University of Batna 2
Faculty of Mathemathics and Informatics

CALCULS DISTRBUES

Cours Pour Master 1 ISIDS

Auteur
Hamouma Moumen

Octobte 1, 2021

Contents

Chapter 1

Introduction

Distributed computing occurs when one has to solve a problem in terms of physically distinct entities (usually called nodes, processors, processes, agents, sensors, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem. In the following, we use the term *process* to denote any computing entity. From an operational point of view this means that the processes of a distributed system need to exchange information, and agree in some way or another, in order to cooperate to a common goal. If processes do not cooperate, the system is no longer a distributed system. Hence, a distributed system has to provide the processes with communication and agreement abstractions.

Understanding and designing distributed applications is not an easy task . This is because, due to the very nature of distributed computing, no process can capture instantaneously the global state of the application it is part of. This comes from the fact that, as processes are geographically localized at distinct places, distributed applications have to cope with the uncertainty created by asynchrony and failures. As a simple example, it is impossible to distinguish a crashed process from a very slow process in an asynchronous system prone to process crashes.

As in sequential computing, a simple approach to facilitate the design of distributed applications consists in designing appropriate abstractions. More generally, computer science is a science of abstraction and distributed computing is the science of distributed abstractions . With such abstractions, the application designer can think about solutions to solve problems at a higher conceptual level than the basic send/receive communication level.

Broadcast abstractions are among the most important abstractions encountered in fault-tolerant distributed computing. Roughly speaking, these abstractions allow processes to disseminate information in such a way that specific provable properties concerning this dissemination are satisfied. One of the most popular of these abstractions is reliable broadcast.

As far as agreement abstractions are concerned, *non-blocking atomic commit* and *consensus* are certainly the most important abstractions of fault-tolerant distributed computing. Assuming that each process proposes a value, the consensus abstraction allows the non-faulty processes to agree on the same value, which has to satisfy some validity condition depending on both the proposed values and the failure model .

This documents presents several protocols that solving Byzantine Consensus with different types of assumptions.

Chapter 2

Signature-Free Asynchronous Binary Byzantine Consensus

2.1 Introduction

This chapter presents a round-based asynchronous consensus algorithm that copes with up to $t < n/3$ Byzantine processes, where n is the total number of processes. In addition of being signature-free and optimal with respect to the value of t, this algorithm has several noteworthy properties: the expected number of rounds to decide is four, each round is composed of two or three communication steps and involves $O(n^2)$ messages, and a message is composed of a round number plus a single bit. To attain this goal, the consensus algorithm relies on a common coin as defined by Rabin, and a new extremely simple and powerful broadcast abstraction suited to binary values. The main target when designing this algorithm was to obtain a cheap and simple algorithm.

2.2 Computation Model

Asynchronous processes The system is made up of a finite set Π of $n > 1$ asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_n\}$. "Asynchronous" means that each process proceeds at its own pace, which may vary arbitrarily with time, and remains always unknown to the other processes.

Communication network The processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that a message that has been sent is eventually received by its destination process, i.e., there is no bound on message transfer delays. "Reliable" means that the network does not loss, duplicate, modify, or create messages. "Point-to-point" means that there is a bi-directional communication channel between each pair of processes. Hence, when a process receives a message, it can identify its sender.

A process p_i sends a message to a process p_j by invoking the primitive "send TAG(m) to p_j", where TAG is the type of the message and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. A process receives a message by executing the primitive "receive()".

The operation broadcast TAG(m) is a macro-operation which stands for "**for each** $j \in \{1, \ldots, n\}$ send TAG(m) to p_j **end for**".This operation is usually called *unreliable* broadcast (if the sender crashes in the middle of the **for** loop, it is possible that only an arbitrary subset correct processes receives the message).

Failure model Up to t processes may exhibit a *Byzantine* behavior. A Byzantine process is a process that behaves arbitrarily: it may crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Hence, a Byzantine process, which is assumed to send a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. Moreover, Byzantine processes can collude to "pollute" the computation. A process that exhibits a Byzantine behavior is called *faulty*. Otherwise, it is *correct*.

Let us notice that, as each pair of processes is connected by a channel, no Byzantine process can impersonate another process. Byzantine processes can modify the message delivery schedule, but cannot affect network reliability.

Notation This computation model is denoted $\mathcal{BZ}\_\mathcal{AS}_{n,t}[\emptyset]$. In the following, this model is both restricted with a constraint on t and enriched with an object providing processes with additional computational power. More precisely, $\mathcal{BZ}\_\mathcal{AS}_{n,t}[n > 3t]$ denotes the model $\mathcal{BZ}\_\mathcal{AS}_{n,t}[\emptyset]$ where the number of faulty processes is smaller than $n/3$, and $\mathcal{BZ}\_\mathcal{AS}_{n,t}[n > 3t, \text{CC}]$ denotes the model $\mathcal{BZ}\_\mathcal{AS}_{n,t}[n > 3t]$ enriched with the common coin (CC) abstraction as defined by Rabin.

2.3 The Binary-Value Broadcast Abstraction

2.3.1 Binary-value broadcast

Definition This communication abstraction (in short, BV-broadcast) in an all-to-all abstraction that provides the processes with a single operation denoted BV_broadcast(). When a process invokes BV_broadcast TAG(m), we say that it "BV-broadcasts the message TAG(m)". The content of a message m is 0 or 1 (hence the term "binary-value" in the name of this communication abstraction).

In a BV-broadcast instance, each correct process p_i BV-broadcasts a binary value and obtains binary values. To store the values obtained by each process, BV-broadcast provides each correct process p_i with a read-only local variable denoted bin_values_i. This variable is a set, initialized to $\emptyset$, which increases when new values are received. VB-broadcast is defined by the four following properties.

- BV-Obligation. If at least $(t + 1)$ correct processes BV-broadcast the same value v, v is eventually added to the set bin_values_i of each correct process p_i.
- BV-Justification. If p_i is correct and $v \in bin\_values_i$, v has been BV-broadcast by a correct process.
- BV-Uniformity. If a value v is added to the set bin_values_i of a correct process p_i, eventually $v \in bin\_values_j$ at every correct process p_j.
- BV-Termination. Eventually the set bin_values_i of each correct process p_i is not empty.

The following property is an immediate consequence of the previous properties.

Property 1. Eventually the sets bin_values_i of the correct processes p_i become non-empty and equal, contain all the values broadcast by correct processes and no value broadcast only by Byzantine processes.

2.3.2 A BV-broadcast algorithm

A simple algorithm implementing the BV-broadcast abstraction is described in Figure 2.1. This algorithm is based on a particularly simple "echo" mechanism. Differently from previous echo-based algorithms , echo is used here with respect to each value that has been received (whatever the number of processes that broadcast it), and not with respect to each pair composed of a value plus the identity of the process that broadcast this value. In the algorithm of Figure 2.1, a value entails a single echo, whatever the number of processes that have broadcast this value.

When a process invokes BV_broadcast MSG(v), $v \in \{0, 1\}$, it broadcasts B_VAL(v) to all the processes (line 01). Then, when a process p_i receives (from any process) a message B_VAL(v), (if not yet done) it forwards this message to all the processes (line 03) if it has received the same message from at least $(t + 1)$ different processes (line 02). Moreover, if p_i has received v from at least $(2t + 1)$ different processes, the value v is added to bin_values_i.

```
operation BV_broadcast MSG(v_i) is
(01)  broadcast B_VAL(v_i).

when B_VAL(v) is received
(02)  if (B_VAL(v) received from (t+1) different processes and B_VAL(v) not yet broadcast)
(03)     then broadcast B_VAL(v)     %  a process echoes a value only once  %
(04)  end if;
(05)  if (B_VAL(v) received from (2t+1) different processes)
(06)     then bin_values_i ← bin_values_i ∪ {v}     %  local delivery of a value  %
(07)  end if.
```

Figure 2.1: An algorithm implementing BV-broadcast in $\mathcal{BZ}\_\mathcal{AS}_{n,t}[n > 3t]$

Remark It is important to notice that no correct process p_i can know when its set bin_values_i has obtained its final value. (Otherwise, consensus will be directly obtained by directing each process p_i to deterministically extract the same value from bin_values_i). This impossibility is due to the net effect of asynchrony and process failures (FLP 85).

2.3.3 Cost of the algorithm

As far as the cost of the algorithm is concerned, we have the following for each BV-broadcast instance.

- If all correct processes BV-broadcast the same value, the algorithm requires a single communication step. Otherwise, it can require two communication steps.
- Let $c \geq n - t$ be the number of correct processes.

The correct processes send $c\ n$ messages if they BV-broadcast the same value, and send $2\ c\ n$ messages otherwise.

- In addition to the control tag B_VAL, a message carries a single bit. Hence, message size is constant.

2.4 The Byzantine Consensus Algorithm

2.4.1 Byzantine consensus and enriched model

Binary Byzantine consensus The Byzantine consensus problem has been informally stated in the Introduction. Assuming that each correct process p_i proposes a value $v_i \in \{0, 1\}$, each of them has to decide a value such that the following property are satisfied.

- BC-Validity. A decided value was proposed by a correct process.
- BC-Agreement. No two correct processes decide different values.
- BC-One-shot. A correct process decides at most once.
- BC-Termination. Each correct process decides.

The BC-Validity property states that no value proposed only by faulty processes can be decided. As we consider binary consensus, it is equivalent to the following property: if all correct processes propose the same value v, the value $\overline{v}$ cannot be decided (where $\overline{v}$ is the other binary value).

Enriching the basic asynchronous model: Rabin's common coin As indicated in the Introduction, the basic system model $\mathcal{BZ}\_\mathcal{AS}_{n,t}[t < n/3]$ has to be enriched so that Byzantine consensus can be solved. The additional computational power we consider here is a *common coin* (CC) as defined by Rabin. As already indicated, the corresponding enriched system model is denoted $\mathcal{BZ}\_\mathcal{AS}_{n,t}[t < n/3, \mathrm{CC}]$. A common coin can be seen as a global entity that delivers the very same sequence of random bits $b_1, b_2, \ldots, b_r, \ldots$ to processes, each bit b_r has the value 0 or 1 with probability 1/2.

More precisely, this oracle provides the processes with a primitive denoted random() that returns a bit each time it is called by a process. In addition to being random, this bit has the following global property: the r^{th} invocation of random() by a correct process p_i returns it the bit b_r. This means that the r^{th}invocations of random() by any pair of correct processes p_i and p_j return them b_r. A common coin is built in such a way that the processes need to cooperate to compute the value of each bit b_r. This is required to prevent Byzantine processes from computing bit values in advance and exploit these values to produce message delivery schedule that would prevent termination.

On randomized consensus When using additional computing power provided by common coins, the consensus termination property can no longer be deterministic. *Randomized consensus* is defined by BC-Validity (Obligation), BC-Agreement, plus the following BC-Termination property : Every non-faulty process decides with probability 1. For round-based algorithms, this termination property is re-stated as follows: For any correct process p_i: $\lim_{r \to +\infty} \big(\text{Probability } [p_i \text{ decides by round } r]\big) = 1$.

2.4.2 Randomized Byzantine consensus algorithm

Principles and description of the algorithm The consensus algorithm is described in Figure 2.2. It requires $t < n/3$ and is consequently optimal with respect to the maximal number t of Byzantine processes that can be tolerated. A process p_i invokes propose(v_i) where $v_i \in \{0, 1\}$ is the value it proposes. It decides when it executes the statement decide(v) at line 08.

The local variable est_i of a process p_i keeps its current estimate of the decision (initially $est_i = v_i$). The processes proceed by consecutive asynchronous rounds and a BV-broadcast instance is associated with each round. The local variable r_i denotes the current round of process p_i, while the local variable $bin\_values_i[r_i]$ denotes the local read-only variable bin_values_i associated with the BV-broadcast instance used at round r_i.

```
operation propose(v_i)
est_i ← v_i; r_i ← 0;
repeat forever
(01)   r_i ← r_i + 1;
(02)   BV_broadcast EST[r_i](est_i);
(03)   wait until (bin_values_i[r_i] ≠ ∅);
       % bin_values_i[r_i] has not necessarily obtained its final value when the wait statement terminates %
(04)   broadcast AUX[r_i](w) where w ∈ bin_values_i[r_i];
(05)   wait until (∃ a set of (n − t) AUX[r_i](x) messages delivered from distinct processes such that
                   values_i ⊆ bin_values_i[r_i] where values_i is the set of values x carried by
                   these (n − t) messages):
(06)   s ← random();
(07)   if (values_i = {v})  % i.e.,|values_i| = 1 %
(08)     then if (v = s) then decide(v) if not yet done end if;
(09)          est_i ← v
(10)     else est_i ← s
(11)   end if
end repeat.
```

Figure 2.2: A BV-broadcast-based algorithm implementing binary consensus in $\mathcal{BZ}\_\mathcal{AS}_{n,t}[n > 3t, \text{CC}]$

The behavior of a correct process p_i during a round r_i can be decomposed in three phases.

- Phase 1: lines 01-03. This first phase is an exchange phase. During a round r_i, a process p_i first invokes BV_broadcast EST$[r_i](est_i)$ (line 02) to inform the other processes of the value of its current estimate est_i. This message is tagged EST and associated with the round number r_i (hence the notation EST$[r_i]()$). Then, p_i waits until its underlying read-only BV-broadcast variable $bin\_values_i[r_i]$ is no longer empty (line 03). Due to the BV-Termination property, this eventually happens. When the predicate becomes satisfied, $bin\_values_i[r_i]$ has not yet necessarily its final value, but it contains at least one value $\in \{0, 1\}$. Moreover, due to the BV-Justification property, the values in $bin\_values_i[r_i]$ were BV-broadcast by correct processes.
- Phase 2: lines 04-05. The second phase is also an exchange phase during which each correct process p_i invokes broadcast AUX$[r_i](w)$ where w is a value that belongs to $bin\_values_i[r_i]$ (line 04). Let us notice that all the correct processes p_j broadcast a

value of their set $bin\_values_j[r_j]$ (i.e., an estimate value of a correct process), while a Byzantine process can broadcast an arbitrary binary value. To summarize, the broadcasts of the second phase inform the other processes of estimate values that have been BV-broadcast by correct processes.

A process p_i then waits until the predicate of line 05 becomes satisfied. This predicate is used to discard values sent only by Byzantine processes. From an operational point of view, it states that there is a set $values_i$ containing only the values broadcast at line 04 by $(n-t)$ distinct processes, and these values originated from correct processes (which BV-broadcast them at line 02). Said in another way, the set $values_i$ of a correct process p_i cannot contain an estimate value broadcast only by Byzantine processes. Hence, after line 05, we have $values_i \in \{0, 1\}$, and for any $v \in values_i$, v is an estimate VB-broadcast by a correct process.

- Phase 3: lines 06-11. This last phase is a local computation phase. A correct process p_i first obtains the common coin value s associated with the current round (line 06).
  - If $|values_i| = 2$, both the value 0 and the value 1 are estimate values of correct processes. In this cases, p_i adopts the value s of the common coin (line 10).
  - If $|values_i| = 1$, p_i decides v (the single value present in $values_i$) if additionally $s = v$ (line 08). Otherwise it adopts v as its new estimate (line 09).

  The statement decide() used at line08 allows the invoking process p_i to decide but does not stop its execution. A process executes round forever. This facilitates the description and the understanding of the algorithm.

Cost of the algorithm As far as the cost of the algorithm is concerned, we have the following, where $c \geq n - t$ denotes the number of correct processes.

- If the correct processes propose the same value, each round requires two communication steps (one in the BV-broadcast and one broadcast), and the expected number of rounds to decide is two. Moreover, the total number of messages sent by correct processes is then $2\ c\ n$.
- If the correct processes propose different values, each round requires three communication steps (two in the BV-broadcast and one broadcast), and the expected number of rounds to decide is four. Moreover, the total number of messages sent by the correct processes is then $4\ c\ n$ per round.
- In addition to a round number, both a message EST[r]() and a message AUX[r]() sent by a correct process carry a single bit. An underlying message B_VAL() has to carry a round number and a bit.
- The total number of bits exchanged by the correct processes is $O(n^2 r \log r)$ where r is the number of rounds executed by the correct processes. Hence, the expected bit complexity is $O(n^2)$.

2.5 Conclusion

This chapter has presented a consensus algorithm suited to asynchronous systems composed of n processes, and where up to $t < n/3$ processes may have a Byzantine behavior. This algorithm relies on Rabin's common coin and an underlying binary-value broadcast algorithm which guarantees that a value broadcast only by Byzantine processes is never

delivered to the correct processes. In addition to being t-resilient optimal, the algorithm, which is round-based and signature-free, uses two or three communication steps per round (this depends on the estimate values of the correct processes at the beginning of a round), and $O(n^2)$ messages per rounds. Moreover, each message carries a round number and a single bit, and the expected number of rounds to decide is four. .

Chapter 3

Randomized k-Set Agreement in Asynchronous Systems

3.1 Introduction

k-Set agreement is a central problem of fault-tolerant distributed computing. Considering a set of n processes, where up to t may commit failures, let us assume that each process proposes a value. The problem consists in defining an algorithm such that each non-faulty process decides a value, at most k different values are decided, and the decided values satisfy some context-depending validity condition. Algorithms solving k-set agreement in synchronous message-passing systems have been proposed for different failure models (mainly process crashes, and process Byzantine failures). Differently, k-set agreement cannot be solved in failure-prone asynchronous message-passing systems when $t \geq k$. To circumvent this impossibility an asynchronous system must be enriched with additional computational power.

Assuming $t \geq k$, this Chapter presents two distributed algorithms that solve k-set agreement in asynchronous message-passing systems where up to t processes may commit crash failures (first algorithm) or more severe Byzantine failures (second algorithm). To circumvent k-set agreement impossibility, this chapter considers that the underlying system is enriched with the computability power provided by randomization. Interestingly, the algorithm that copes with Byzantine failures is signature-free, and ensures that no value proposed only by Byzantine processes can be decided by a non-faulty process. Both algorithms share basic design principles.

3.2 Asynchronous Model with Crashes Failures, and Definitions

3.2.1 Computation Model

Asynchronous processes The system is made up of a finite set Π of $n > 1$ asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_n\}$. "Asynchronous" means that each process proceeds at its own pace, which may vary arbitrarily with time, and remains always unknown to the other processes.

Communication network The processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that a message is eventually received by its destination process, i.e., there is no bound on message transfer delays. "Reliable" means that the network does not loss, duplicate, modify, or create messages. "Point-to-point" means that there is a bi-directional communication channel between each pair of processes. Hence, when a process receives a message, it can identify its sender.

A process p_i sends a message to a process p_j by invoking the primitive operation send TAG(m) to p_j, where TAG is the type of the message and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. A process receives a message by executing the primitive "receive()".

The operation broadcast TAG(m) is a macro-operation which stands for "**for each** $j \in \{1, \ldots, n\}$ send TAG(m) to p_j **end for**". This operation is usually called *unreliable* broadcast (if the sender crashes in the middle of the **for** loop, it is possible that only an arbitrary subset correct processes receives a message).

Failure model Up to t processes may crash during an execution. As already indicated in the Introduction, before a process (possibly) crashes, it executes its code as defined by its local algorithm, and no crashed process recover. A crash is consequently a definitive halting.

Given an execution, a process that crashes is said to be *faulty* in this execution, otherwise it is *correct* or *non-faulty*. Hence, before a process crashes, no one knows if it correct or faulty.

Random multi-sided local coin Each process p_i is endowed with an operation denoted random(). Each invocation of this operation takes a non-empty set X as input parameter and returns a value of X with probability $1/|X|$. As we will see in Section 3.3, equipping each process with such a local random coin provides an additional computational power that allows k-set agreement to be solved.

Notation This computation model is denoted $[\emptyset]$ (CAMP stands for "Crash-prone Asynchronous Message Passing"). In the following, this model is both restricted with a constraint on t and enriched with random multi-sided local coins, which provide the processes with additional computational power. More precisely, $[t < n/\alpha]$ (where α is a positive integer) denotes the model $[\emptyset]$ where the maximal number of faulty processes is smaller than n/α. $[t < n/\alpha, \text{LRC}]$ denotes the model $[t < n/\alpha]$ where each process is enriched with a local multi-sided random coin. Let us notice that, as LRC belongs to the model, it is given for free in $[t < n/\alpha, \text{LRC}]$.

Time complexity When computing the time complexity we ignore local computation time, and consider the longest sequence of causally relate messages m_1, m_2, ..., m_z (i.e., for any $x \in [2..z]$, the reception of m_{x-1} is a requirement for the sending of m_x). The size of such a longest sequence defines the time complexity.

3.2.2 Reliable broadcast abstraction

This communication abstraction (in short R-Broadcast) provides the processes with two operations, denoted R_broadcast() and R_deliver(). When a process invokes R_broadcast TAG(m), we say that it "r-broadcasts" the message whose type is TAG and value is m. Similarly, when a process returns from the invocation of R_deliver() we say that it "r-delivers" a message. Reliable broadcast is defined by the following properties.

- R-Validity. It a process r-delivers TAG(m) from a process p_j, p_j invoked R_broadcast TAG(m).
- R-Integrity. A process r-delivers at most once a message TAG(m) from a sender p_i.
- R-Termination. If a correct process r-broadcasts a message TAG(m), or a correct process r-delivers the message TAG(m), then all correct processes r-deliver the message TAG(m).

Validity relies the outputs to the inputs (no spurious messages). Assuming no process r-broadcasts several times the same message (which can be easily implemented by associating a new sequence number with each message r-broadcast by a process), Integrity states there is no duplication. Finally, Termination states the conditions under which a message must be r-delivered by all correct processes, namely, either when its sender is correct, or when at least one correct process r-delivered it.

It is easy to see that, all correct processes r-deliver the same set of messages M, and this set contains all the messages they r-broadcast. Moreover, a faulty process r-delivers subset of M, but two faulty processes can r-deliver two sets of messages $M1$ and $M2$ such that none of $M1$ and $M2$ contains the other set.

Implementations of R-Broadcast can be easily designed in [$\emptyset$]. A very simple (but inefficient) one is the following. When, at the implementation level, a process receives for the first time a copy of the message TAG(m), it first forwards it to all the other processes, and only then r-delivers it. According to the underlying topology and the way message identifiers are built, more efficient implementations can be designed .

3.2.3 k-Set agreement

The k-agreement problem was introduced in the context of the model [$\emptyset$]. It consists in implementing an operation denoted propose$_k$() satisfying the properties stated below. This operation takes an input parameter, and returns a value. When a process invokes propose$_k(v)$, we say that it "proposes value v". When a process returns from propose$_k$() with the value w, we say that it "decides w". It is assumed that at least the correct processes invoke propose$_k$(). The properties defining k-set agreement are the following.

- C-KS-Validity. It a process decides v, there is a process that proposed v.
- C-KS-Agreement. At most k different values are decided.
- C-KS-Termination. Any correct process decides a value.

As before, Validity relies the outputs to the inputs. Agreement defines a coordination constraint on the processes. Termination states that at least the processes that do not crash decide.

3.3 Crash Model: A Randomized k-Set Agreement Algorithm

This section presents an algorithm which solves the k-set agreement problem in $[t < n/2, \text{LRC}]$. This algorithm is a round-based algorithm, which means that the processes execute a sequence of asynchronous rounds[1].

As we are interested in a randomized algorithm to solve k-set agreement, the Termination property is weakened as follows : any correct process decides with probability 1. In the context of round-based algorithms, this property can be re-stated as follows, where p_i is any correct process:

$$\text{C-KS-P-Termination:} \quad \lim_{r \to +\infty} \big(\text{Probability } [p_i \text{ decides by round } r]\big) = 1.$$

3.3.1 Description of the algorithm

Each process p_i starts the algorithm by invoking $\mathsf{propose}_k(v_i)$, where v_i is the value it proposes. It decides a value when it executes the statement $\mathsf{return}(v)$; v is then the value it decides. Moreover, when it executes $\mathsf{return}()$, a process terminates its participation to the algorithm. $\perp$ denotes a default value that no process can propose. It is used during each round to restrict the set of proposed values to a set of at most k values.

The algorithm is described in Figure 3.1. Each process manages a local variable est_i, which represents the current estimate of its decision value. Initially, est_i is set to v_i (the value proposed by p_i). Process p_i manages also a local array $val_i[1..n]$, initialized to $[\perp, ..., \perp]$.

Dissemination of the proposed values When, it starts, a process p_i first r-broadcasts the value it proposes (line 01). When, it r-delivers the value proposed by p_j, p_i saves it in $val_i[j]$ (line 16). Let us notice that, due to the Validity and Termination properties of R-broadcast, the arrays $val[1..n]$ of the correct processes eventually (a) contain at least the values proposed by each correct process, and (b) become equal.

A sequence of asynchronous rounds The processes execute a sequence of asynchronous rounds to converge to a set of at most k values. Each round is made up of two communication phases (hence it costs two communication steps). The aim of the first phase (lines 03-06) is to force each process to adopt either a value from a set of at most k different values, or the default value $\perp$.The aim of the second phase (lines 07-13) is to allow processes to decide non-$\perp$ values that have been previously adopted, while ensuring that (if processes decide during distinct rounds) no more than k different values will eventually be decided (i.e., the Agreement property is not violated).

Let us notice that, differently from the R-broadcast used at lines 01 and 11, the broadcast operation used at lines 03 and 07 is the unreliable macro-operation defined in Section 3.2.1.

[1]Differently from round-based synchronous algorithms where the progress from a round to the next one is a built-in property provided by the model, in an asynchronous system it is to the processes to implement the progress of a round to the next one.

```
operation propose_k(v_i) is
(01)  val_i ← [⊥,...,⊥]; r_i ← 0; est_i ← v_i; R_broadcast VAL(v_i);
(02)  while true do r_i ← r_i + 1; % round r_i = r %
// ————————- phase 1 of round r_i: From up to n values to up to k values plus possibly ⊥ ———-
(03)    broadcast PHASE1(r_i, est_i);
(04)    wait (PHASE1(r_i, −) received from R = k⌊n/(k+1)⌋ + 1 processes);
(05)    if (∃v | W = ⌊n/(k+1)⌋ + 1 PHASE1(r_i, v) messages have been received)
(06)                                then ph2_est_i ← v else ph2_est_i ← ⊥ end if;
// ————————- phase 2 of round r_i: Try to decide on one of at most k values ——————————
(07)    broadcast PHASE2(r_i, ph2_est_i);
(08)    wait (PHASE2(r_i, ph2_est) received from maj = ⌊n/2⌋ + 1 processes);
(09)    let ph2_rec_i= { ph2_est such that PHASE2(r_i, ph2_est) has been received };
(10)    case ph2_rec_i = {⊥}          then est_i ← val_i[random([1..n])]
(11)         ⊥ ∉ ph2_rec_i            then let v be any value ∈ ph2_rec_i; R_broadcast DEC(r_i, v)
(12)         ph2_rec_i = {⊥, v, ...}  then est_i ← any non-⊥ value ∈ ph2_rec_i
(13)    end case
(14)  end while.

(15)  when VAL(v) is r-delivered from p_j do val_i[j] ← v.

(16)  when DEC(r, v) is r-delivered from p_j do return(v).
```

Figure 3.1: Solving k-set agreement in $[t < n/2, \text{LRC}]$

First phase of a round r The processes first exchange their current estimate values (lines 03-04). Let us note that, as far the round r is concerned, a message PHASE1(r, v) can be interpreted as a vote for the value v. Accordingly, a process p_i adopts a value if has received enough votes for it, say W votes. If, among the values it has received, none has enough votes to be adopted, p_i adopts the default value $\perp$. The adopted value is kept in $ph2\_est_i$ (line 06).

The aim is to have at most k different values adopted by the processes at the end of the first phase. In order to attain this goal, we must have $(k+1)W > n$ (as there are only n processes, $k+1$ values cannot each obtain W votes). This means that $W = \lceil \frac{n+1}{k+1} \rceil = \lfloor \frac{n}{k+1} \rfloor + 1$.

Let us now examine how many messages PHASE1(r, v) a process has to wait for (at line 04) before adopting a value (line 06) in order to have a chance to adopt a value initially proposed by a process (i.e., a value different from $\perp$). Let R be this number. Considering the case where p_i adopts a non-$\perp$ value, let us examine the worst situation: p_i can receive $(W-1)$ votes for $(k-1)$ different values, and only then receive W votes for the value v it adopts. Hence, $R = (W-1)(k-1)+W = (W-1)k+1$. Moreover, in order that no process blocks at line 04, we must have $R \leq n - t$ which is equivalent to $t < n - k\lfloor \frac{n}{k+1} \rfloor$.

Hence, at the end of the first phase, the set of the local variables $ph2\_est_i$ contains at most k values, plus possibly $\perp$. The aim of the second phase is to allow each process to decide one of these non-$\perp$ values in such a way that the Agreement property be not violated even if processes decide during different rounds.

Second phase of a round r During the second phase, the processes exchange the values they have previously adopted. A process p_i waits for messages PHASE2() from a majority of processes (lines 07-08). As shown at line 09, $ph2\_rec_i$ is the set of values

received by p_i. Let us notice that if v ($\neq \perp$) belongs to $ph2\_rec_i$, then v was the estimate of at most W processes at the beginning of the current round. There are three cases determined by the content of $ph2\_rec_i$.

- If $\perp \notin ph2\_rec_i$, p_i can decide any value v of this set (line 11). It then r-broadcasts the message DEC(v). If p_i does not crash, this message will be r-delivered at all the non-crashed processes, which (if they do not have yet decided) will decide v at line 15.
- If $ph2\_rec_i$ contains both $\perp$ and non-$\perp$ values, p_i updates its estimate est_i to any non-$\perp$ value of $ph2\_rec_i$, and proceeds to the next round.
- If $ph2\_rec_i$ contains only the default value $\perp$, p_i updates its current estimates est_i to a randomly chosen value (line 10), and then proceeds to the next round. Actually, p_i selects randomly a process identity (say x) and sets est_i to $val_i[x]$. Let us note that $val_i[x]$ is equal to the value proposed by p_x or $\perp$. The randomness of the choices guarantees that eventually there are rounds during which p_i selects non-$\perp$ entries of its array $val_i[1..n]$.

It is important to observe that, as soon as a process returned from the R-broadcast of line 11, all correct processes will eventually return a value. Said, differently, no deadlock is possible as soon as a process has executed line 11.

3.4 Asynchronous Model with Byzantine Failures, and Definitions

3.4.1 Computation Model

From to Byzantine failures The computation model is the asynchronous message passing model presented in Section 3.2 enriched with local random coins (LRC). It differs only in the nature of process failures.

Failure model Up to t processes may exhibit a *Byzantine* behavior . A process that exhibits a Byzantine behavior is called *faulty*. Otherwise, it is *correct* or *non-faulty*. A Byzantine process is a process that behaves arbitrarily: it may crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. As a simple example, a Byzantine process, which is assumed to send a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the other processes. More generally, a Byzantine process has an unlimited computational power, and Byzantine processes can collude to "pollute" the computation. Let us notice that, as each pair of processes is connected by a channel, no Byzantine process can impersonate another process, but Byzantine processes are not prevented from influencing the delivery order of messages sent to correct processes.

Discarding messages from Byzantine processes If, according to its algorithm, a process p_j is assumed to send a single message TAG() to a process p_i, then p_i processes only the first message TAG(v) it receives from p_j. This means that, if p_j is Byzantine and sends several messages TAG(v), TAG(v') where $v' \neq v$, etc., all of them except the first

one are discarded by their receivers. (Let us observe that this does not prevent multiple copies of the first message TAG() to be received and processed by their receiver.)

Notation This computation model is denoted $[\emptyset]$ (BAMP stands for "Byzantine Asynchronous Message Passing"). As for $[\emptyset]$, this basic model is both restricted with a constraint on t and enriched with local coins. It is consequently denoted $[t < n/\alpha, \text{LRC}]$, where $\alpha \geq 1$.

3.4.2 The no-duplicity broadcast abstraction

The foll wing broadcast abstraction will be a basic component used in the all-to-all SMV-broadcast abstraction presented in Section 3.5 (which is the communication abstraction on which is built the Byzantine-tolerant k-set algorithm presented in Section 3.6).

Definition of the ND-broadcast communication abstraction This abstraction is defined by two operations denoted ND_broadcast() and ND_deliver(), which allow the processes to eliminate bad behaviors of Byzantine processes. More precisely, a Byzantine process is prevented from sending different messages to different correct processes, while it is assumed to send the very same message to all of them.

As previously, when a process invokes ND_broadcast TAG() we say that it "ND-broadcasts" a message, and when it invokes ND_deliver() we say that it "ND-delivers" a message. Considering an instance of ND-broadcast where the operation ND_broadcast TAG() is invoked by a process p_i, this communication abstraction is defined by the following properties.

- ND-Validity. If a non-faulty process ND-delivers a message from p_i, then, if it is non-faulty, p_i ND-broadcast this message.
- ND-No-duplicity. No two non-faulty processes ND-deliver distinct messages from p_i.
- ND-Termination. If the sender p_i is non-faulty, all the non-faulty processes eventually ND-deliver its message.

Let us observe that, if the sender p_i is faulty, it is possible that some non-faulty processes ND-deliver a message from p_i while others do not ND-deliver a message from p_i. As already indicated, the no-duplicity property prevents non-faulty processes from ND-delivering different messages from a faulty sender.

An algorithm implementing ND-broadcast It is shown that $t < n/3$ is a necessary requirement to implement ND-broadcast in a Byzantine asynchronous message-passing system. Algorithm 3.2 implements ND-broadcast in ${}_{n,t}[t < n/3]$.

When a process p_i wants to ND-broadcast a message whose content is v_i, it broadcasts the message ND_INIT(i, v_i) (line 01). When a process receives a message ND_INIT$(j, -)$ for the first time, it broadcasts a message ND_ECHO(j, v) where v is the data content of the ND_INIT() message (line 02). If the message ND_INIT(j, v) received is not the first message ND_INIT$(j, -)$, p_j is Byzantine and the message is discarded. Finally, when p_i has received the same message ND_ECHO(j, v) from $(n - t)$ different processes, it locally ND-delivers MSG(j, v) (lines 03-04).

```
operation ND_broadcast MSG(v_i) is
(01)  broadcast ND_INIT(i, v_i).

when ND_INIT(j, v) is delivered do
(02)  if (first reception of ND_INIT(j, −)) then broadcast ND_ECHO(j, v) end if.

when ND_ECHO(j, v) is delivered do
(03)  if (ND_ECHO(j, v) received from (n − t) different processes and MSG(j, v) not yet ND_delivered)
(04)    then ND_deliver MSG(j, v)
(05)  end if.
```

Figure 3.2: Implementing ND-broadcast in $[t < n/3]$ (Algorithm 1)

The algorithm considers an instance of ND-broadcast, i.e., a correct process invokes at most once ND-broadcast. Adding a sequence number to each message allows any process to ND-broadcast a sequence of messages.

Theorem 1. Algorithm *3.2* implements *ND-broadcast* in the system model $[t < n/3]$.

Proof (The proof is from **[T84]**. It is given for completeness.) To prove the ND-termination property, let us consider a non-faulty process p_i that ND-broadcasts the message $\text{MSG}(v_i)$. As p_i is non-faulty, the message $\text{ND\_INIT}(i, v_i)$ is received by all the non-faulty processes, which are at least $(n - t)$, and every non-faulty process broadcasts $\text{ND\_ECHO}(i, v_i)$ (line 02). Hence, each non-faulty process receives the message $\text{ND\_ECHO}(i, v_i)$. from $(n - t)$ different processes. It follows that every non-faulty process eventually ND-delivers the message $\text{MSG}(i, v_i)$ (lines 03-04).

To prove the ND-no-duplicity property, let us assume by contradiction that two non-faulty processes p_i and p_j ND-deliver different messages m_1 and m_2 from some process p_k (i.e., $m_1 = \text{MSG}(k, v)$ and $m_2 = \text{MSG}(k, w)$, with $v \neq w$). It follows from the predicate of line 03, that p_i received $\text{ECHO}(k, v)$ from a set of $(n - t)$ distinct processes, and p_j received $\text{ECHO}(k, w)$ from a set of $(n - t)$ distinct processes. As $n > 3t$, it follows that the intersection of these two sets contains a non-faulty process. But, as it is non-faulty, this sent the same message $\text{ND\_ECHO}()$ to p_i and p_j (line 02). Hence, $m_1 = m_2$, which contradicts the initial assumption.

To prove the ND-validity property, we show that, if Byzantine processes forge and broadcast a message $\text{ND\_ECHO}(i, w)$ such that p_i is correct and has never invoked ND_broadcast $\text{MSG}(w)$, then no correct process can ND-deliver $\text{MSG}(i, w)$. Let us observe that at most t processes can broadcast the message $\text{ND\_ECHO}(i, w)$. As $t < n - t$, it follows that the predicate of line 03 can never be satisfied at a correct process. Hence, if p_i is correct, no correct process can ND-deliver from p_i a message that was not been ND-broadcast by p_i.

$\square_{Theorem\ 1}$

It is easy to see that this implementation uses two consecutive communication steps and $O(n^2)$ underlying messages ($n - 1$ in the first communication step, and $n(n - 1)$ in the second one). Moreover, there are two types of protocol messages, and the size of the control information added to a message is $\log_2 n$ (sender identity).

3.4.3 k-Set agreement

Definition The intrusion-tolerant Byzantine (ITB) k-set agreement was informally presented in the introduction. When considering round-based randomized k-set agreement algorithms (namely, the system model [LRC]) these properties are the following.

- B-KS-Validity. If a correct process decides v, then v was proposed by a correct process.
- B-KS-Agreement. The set of values decided by the correct processes contains at most k values.
- B-KS-P-Termination. $\lim_{r\to+\infty}\big(\text{Probability }[p_i \text{ decides by round } r]\big) = 1$.

Additional constraint As stated in the introduction, we assume $k \leq t$. Moreover, we have also seen that, in order for a correct process to decide neither a value proposed only by Byzantine processes, nor a predefined default value, it is assumed that, whatever the domain of the values that can be proposed by the correct processes, in any execution, at most m different values are proposed by correct processes, where m depends on n and t, namely, $n > t(m+1)$. As shown that, this condition is necessary.

Hence, assuming the non-triviality conditions $k \leq t$, and the fact that, in any execution, at most m different values are proposed by the correct processes, the system model considered here to solve the ITB k-set agreement problem is $[t < n/(m+1), \text{LRC}]$.

3.5 Two Multivalued Validated Broadcast Abstractions

This section presents the all-to-all communication abstractions MV-broadcast and SMV-broadcast. "All-to-all" mean that it is assumed that all the non-faulty processes invoke the corresponding broadcast operation. As indicated in the introduction, these abstractions extend to the "multivalue" case the BV-broadcast and SBV-broadcast communication abstractions , which consider binary values only.

3.5.1 Multivalued validated all-to-all broadcast

Definition of MV-broadcast This communication abstraction provides the processes with a single operation denoted MV_broadcast(). When a process invokes MV_broadcast TAG(m), we say that it "MV-broadcasts the message typed TAG and carrying the value m". The invocation of MV_broadcast TAG(m) does not block the invoking process. The aim of MV-broadcast is to eliminate the values (if any) that have been broadcast only by Byzantine processes.

In each instance of the MV-broadcast abstraction, each correct process p_i MV-broadcasts a value and eventually obtains a set of values. To store these values, MV-broadcast provides each process p_i with a read-only local variable denoted mv_values_i. This set variable, initialized to $\emptyset$, increases asynchronously when new values are received. Each instance of MV-broadcast is defined by the four following properties.

- MV-Termination. The invocation of MV_broadcast() by a correct process terminates.
- MV-Justification. If p_i is a correct process and $v \in mv\_valid_i$, v has been MV-broadcast by a correct process.

- MV-Uniformity. If p_i is a correct process and $v \in mv\_valid_i$, eventually $v \in mv\_valid_j$ at every correct process p_j.
- MV-Obligation. Eventually the set mv_valid_i of each correct process p_i is not empty.

The following properties are immediate consequences of the previous definition.

- MV-Equality. The sets mv_valid_i of the correct processes are eventually non-empty and equal.
- MV-Integrity. The set mv_valid_i of a correct process p_i never contains a value MV-broadcast only by Byzantine processes.

On the feasibility condition $n > (m+1)t$ Let m be the number of different values MV-broadcast by correct processes. It follows from the previous specification that, even when the (at most) t Byzantine processes propose the same value w, which is not proposed by correct processes, w cannot belong to the set mv_valid_i of a correct process p_i. This can be ensured if and only if there is a value MV-broadcast by at least $(t+1)$ correct processes. This feasibility condition is captured by the predicate $n - t > mt$. Hence $n > (m+1)t$ is a feasibility condition for MV-broadcast to cope with up to t Byzantine processes. Let us notice that, as $m \geq 2$, $n > (m+1)t$ implies $n > 3t$.

An MV-broadcast algorithm Algorithm 3.3 describes a simple implementation of MV-broadcast, suited to the system model $[t < n/(m+1)]$. This algorithm is based on a simple "echo" mechanism. Differently from previous echo-based algorithms , the echo is used here with respect to each value that has been received (whatever the number of processes that broadcast it), and not with respect to each pair composed of a value plus the identity of the process that broadcast this value. Hence, a value entails at most one echo per process, whatever the number of processes that MV-broadcast this value.

```
let witness(v) = number of different processes from which MV_VAL(v) was received.

operation MV_broadcast MSG(v_i) is
(01)  broadcast MV_VAL(v_i); return().

when MV_VAL(v) is received
(02)  if (witness(v) ≥ t + 1) ∧ (MV_VAL(v) not yet broadcast)
(03)    then broadcast MV_VAL(v)    % a process echoes a value only once %
(04)  end if;
(05)  if (witness(v) ≥ n − t) ∧ (v ∉ mv_valid_i)
(06)    then mv_valid_i ← mv_valid_i ∪ {v}    % local delivery of a value %
(07)  end if.
```

Figure 3.3: Implementing MV-broadcast in $[t < n/(m+1)]$ (Algorithm 2)

When a process p_i invokes MV_broadcast MSG(v_i), it broadcasts MV_VAL(v_i) to all the processes (line 01). Then, when a process p_i receives (from any process) a message MV_VAL(v), (if not yet done) it forwards this message to all the processes (line 03) if it has received the same message from at least $(t+1)$ different processes (line 02). Moreover, if p_i has received v from at least $(2t+1)$ different processes, the value v is added to mv_valid_i (lines 05-06). Let us notice that, except in the case where $|mv\_valid_i| = m$, no correct process p_i can know if its set mv_valid_i has obtained its final value.

Cost of the algorithm As at most m values are MV-broadcast by the correct processes, it follows from the text of the algorithm that each correct process broadcasts each of these values at most once (at line 01 or line 03). Hence, if there are $c \in [n-t..n]$ correct processes, their broadcasts entail the sending of at most m c n messages MV_VAL(). Finally, whatever the number of values that are MV-broadcast, the algorithm requires at most two communication steps.

3.5.2 Synchronized multivalued validated all-to-all broadcast

Definition of SMV-broadcast This all-to-all communication abstraction provides the processes with a single operation denoted SMV_broadcast TAG(). As indicated by its name, its aim is to synchronize processes so that, if a single value v is delivered to a correct process, then v is delivered to all the correct processes.

In each instance of the SMV-broadcast abstraction, each correct process invokes SMV_broadcast TAG(). Such an invocation returns to the invoking process p_i a set denoted $view_i$ and called a local view. We say that a process *contributes* to a set $view_i$ if the value it SMV-broadcasts belongs to $view_i$. SMV-broadcast is defined by the following properties.

- SMV-Termination. The invocation of SMV_broadcast TAG() by a correct process terminates.
- SMV-Obligation. The set $view_i$ returned by a correct process p_i is not empty.
- SMV-Justification. If p_i is correct and $v \in view_i$, then a correct process SMV-broadcast v.
- SMV-Inclusion. If p_i and p_j are correct processes and $view_i = \{v\}$, then $v \in view_j$.
- SMV-Contribution. If p_i is correct, at least $(n-t)$ processes contribute to its set $view_i$.
- SMV-No-duplicity. Let be the union of the sets $view_i$ of the correct processes. A process contributes to at most one value of .

The following property is an immediate consequence of the previous definition. property.

- SMV-Singleton. If p_i and p_j are correct, $[(view_i = \{v\}) \wedge (view_j = \{w\})] \Rightarrow (v = w)$.

Let $v \in$, p_i a correct process, and p_j a Byzantine process. It is possible that, while the value v was SMV-broadcast by p_i (hence p_i contributed to), p_j also appears as contributing to with the same value v. The SMV-No-duplicity property states the following: no value $w \in \setminus\{v\}$ appears as a contribution of p_j.

An SMV-broadcast algorithm Algorithm 3.4 implements the SMV-broadcast abstraction in the system model $[t < n/(m+1)]$. A process p_i first MV-broadcasts a message MSG (v_i) and waits until the associated set mv_values_i is not empty (lines 01-02). Let us remind that, when p_i stops waiting, mv_values_i has not necessarily obtained its final value. Then, p_i extracts a value w from mv_values_i and ND-broadcasts it to all (line 03). Let us notice that, due to the ND-no-duplicity property, no two correct processes can ND-deliver different values from the same Byzantine process.

```
operation SMV_broadcast MSG (v_i) is
(01)  MV_broadcast MSG(est_i);
(02)  wait (mv_values_i ≠ ∅);
      % mv_values_i has not necessarily its final value when the wait statement terminates %
(03)  ND_broadcast ND_AUX(w) where w ∈ mv_values_i;
(04)  wait (∃ a set view_i such that its values (i) belong to mv_values_i, and
            (ii) come from messages ND_AUX() received from (n − t) distinct processes);
(05)  return (view_i).
```

Figure 3.4: Implementing SMV-broadcast in $[t < n/(m+1)]$ (Algorithm 3)

Finally, p_i waits until the predicate of line 04 is satisfied. This predicate has two aims. The first is to discard from $view_i$ (the set returned by p_i) a value broadcast only by Byzantine processes. Hence the predicate $view_i \subseteq mv\_values_i$. The second aim is to ensure that, if the view $view_i$ of a correct process p_i contains a single value, then this value eventually belongs to the view $view_j$ of any correct process p_j. To this end, $(n-t)$ different processes (hence, at least $(n-2t)$ correct processes) must contribute to $view_i$.

Multiset version of SMV-broadcast While a value belongs or does not belong to a set, a multiset (also called a bag) is a set in which the same value can appear several times. As an example, while $\{a, b, c\}$ and $\{a, b, b, c, c, c\}$ are the same set, they are different multisets.

It is easy to see that the "set" version of the SMV-broadcast (where $view_i$ is a set) and Algorithm 3.4 can be easily converted into a "multiset" version where $view_i$ is a multiset.

3.6 Byzantine Model: a Randomized k-Set Agreement Algorithm

This section presents and proves correct an algorithm which the k-set agreement problem in $[t < n/(m+1), \text{LRC}]$. This algorithm is built in a modular way on top of the SMV-broadcast communication abstraction.

3.6.1 Description of the algorithm

Local variables To solve the ITB k-set agreement problem, Algorithm 3.5, which is round-based, relies on a very modular construction. Each process p_i manages two local variables whose scope is the whole execution: a local round number r_i, and a local estimate of a decision value, denoted est_i. It also manages three local variables whose scope is the current round r: a multiset $view_i[r, 1]$, an auxiliary variable aux, and a set $view_i[r, 2]$.

Description of the algorithm When p_i invokes $\mathsf{propose}_k(v_i)$ it assigns v_i to est_i and initializes r_i to 0 (line 01). Then p_i enters a loop that it will exit at line 08 by executing $\mathsf{return}(v)$, which returns the decided value v and stops its participation in the algorithm.

```
operation propose_k(v_i) is
(01) est_i ← v_i; r_i ← 0;
(02) repeat forever
(03)    r_i ← r_i + 1;
// ——————————————— phase 1 ———————————————
(04)    view_i[r_i, 1] ← SMV_broadcast PHASE[r_i, 1](est_i);    % view_i[r_i, 1] is a multiset %
(05)    if (∃v appearing W times in view_i[r_i, 1]) then aux ← v else aux ← ⊥ end if;
// ——————————————— phase 2 ———————————————
(06)    view_i[r_i, 2] ← SMV_broadcast PHASE[r_i, 2](aux);    % view_i[r_i, 2] is a set %
(07)    case (⊥ ∉ view_i[r_i, 2])              then let v be any value ∈ view_i[r_i, 2];
(08)                                                broadcast DECIDE(v); return(v)
(09)          (view_i[r_i, 2] = {⊥, v, ···})   then est_i ← any value non-⊥ ∈ view_i[r_i, 2]
(10)          (view_i[r_i, 2] = {⊥})           then est_i ← random(mv_valid_i[1, 1])
(11)    end case
(12) end repeat.
```

Figure 3.5: Byzantine k-set agreement based on SMV-broadcast, and local random coins (Algorithm 4)

Each round r executed by a process p_i is made up of two phases. During the first phase of round r, each correct process p_i invokes SMV_broadcast(est_i) (multiset version) and stores the multiset returned by this invocation in $view_i[r, 1]$. Let us remind that this multiset contains only values SMV-broadcast by at least one correct process. The aim of this phase is to build a global set [2], denoted $[r]$, which contains at most $(k + 1)$ values, such that at most k of them are contributed by correct processes, and the other one is the default value $\perp$. To this end, each correct process p_i checks if there is a value v that appears "enough" (say W) times in the multiset $view_i[r, 1]$. If there is such a value v, p_i adopts it (assignment $aux \leftarrow v$), otherwise it adopts the default value $\perp$ (line 05).

The set $[r]$ is made up of the aux variables of all the correct processes. For $[r]$ to contain at most k non-$\perp$ values, W has to be such that $(k+1)W > n$ (there are not enough processes for $(k+1)$ different values such that each of them was contributed by W processes. Hence, $W > n/(k + 1)$.

When it starts the second phase of round r, each correct process p_i invokes SMV_broadcast(aux) (set version) and stores the set it obtains in $view_i[r, 2]$. Due to the properties of SMV-broadcast, $view_i[r, 2]$ is a local approximation of $[r]$, namely, we have $view_i[r, 2] \subseteq [r]$. Then, the behavior of p_i depends on the content of the set $view_i[r, 2]$.

- If $\perp \notin view_i[r, 2]$, p_i decides any value in $view_i[r, 2]$ (lines 07-08).
- If $view_i[r, 2]$ contains $\perp$ and non-$\perp$ values, p_i updates its current estimate est_i to any non-$\perp$ value of $view_i[r, 2]$ and starts new round (line 09).
- If $view_i[r, 2]$ contains only $\perp$, p_i starts a new round, but updates previously its current estimate est_i to a random value (line 10). This random value is obtained from the set (denoted $mv\_valid_i[1, 1]$ in the algorithm) locally output by the first MV-broadcast instance invoked by p_i. The use of these sets allows the algorithm to benefit from the fact that these sets are eventually equal at all correct processes (MV-Equality property). The KS-Termination relies on this property.

[2] While the value of this set could be known by an external global observer, its value can never be explicitly known by a correct process. However, a process can locally build an approximation of it during the second phase, see below.

As shown in the proof, an important behavioral property of the algorithm lies in the fact that, at any round r, it is impossible for two correct processes p_i and p_j to be such that $(\perp \notin view_i[r,2]) \wedge (view_i[r,2] = \{\perp\})$. These two predicates are mutually exclusive.

On the value of W (This discussion is similar to the one on the definition of W and R appearing in Section 3.3.1.) The value W is used at line 05 for a safety reason, namely, no more than k non-$\perp$ values can belong to the set $[r]$. As we have seen, this is captured by the constraint $W(k+1) > n$. It appears that W has also to be constrained for a liveness reason, namely, when the correct processes start a new round r with at most k different estimates values, none of them must adopt the value $\perp$ at line 05 (otherwise, instead of deciding at line 07, they could loop forever).

This liveness constraint is as follows. Let us consider the size of the multiset $view_i[r,1]$ obtained at line 04. In the worst case, when the correct processes start a new round r with at most k different estimates, $view_i[r,1]$ may contain $(k-1)$ different values, each appearing $(W-1)$ times, and only one value that appears W times. Hence, $view_i[r,1]$ must contain at least $R = (W-1)(k-1) + W = (W-1)k + 1$ elements. As it follows from Algorithm 3.4 that $|view_i[r,1] \geq n - t$, we obtain the liveness constraint $n - t \geq (W-1)k + 1$.

On message identities The messages PHASE() SVM-broadcast at line 04 and line 06 are identified by a pair $[r,x]$ where r is a round number and $x \in \{1,2\}$ a phase number. Each of these messages gives rise to underlying messages ND_AUX() (Algorithm 3.3), MV_VAL() (Algorithm 3.2), and underlying sets *witness*() (Algorithm 3.2). Each of them inherits the pair identifying the message PHASE() it originates from.

On the messages DECIDE() Before a correct process decides a value v, it sends a message DECIDE(v) to each other process (line 08). Then, it stops its execution. This halting has not to prevent correct processes from terminating, which could occur if they wait forever underlying messages ND_AUX() or MV_VAL() from p_i.

To this end, a message DECIDE(v) has to be considered as representing an infinite set of messages. More precisely if, while executing a round r, a process p_i receives a message DECIDE(v) from a process p_j, it considers that it has received from p_j the following set of messages: $\{$ND_AUX$[r',1](v)$, ND_AUX$[r',2](v)$, MV_VAL$[r',1](v)$, MV_VAL$[r',2](v)\}_{r' \geq r}$. It is easy to see that the messages DECIDE() simulate a correct message exchange that could be produced, after it has decided, by a deciding but non-terminating process.

Another solution would consist in using a Reliable Broadcast abstraction that copes with Byzantine processes. In this case, a process could decide a value v as soon as it has RB-delivered $(t+1)$ messages DECIDE(v).

3.7 Conclusion

This chapter was on k-set agreement in two types of asynchronous message-passing, the ones where processes may commit crash failures, and the ones where they may commit Byzantine failures. As k-set agreement cannot be solved in these basic system models without additional computational power, the chapter considered the computational power provided by local multi-sided random coins.

Chapter 4

Implementing Timely Provable reliable Send Primitive

4.1 Introduction

Broadcast abstractions are among the most important abstractions required to address fault-tolerant distributed computing. Roughly speaking, these abstractions allow processes to disseminate information in such a way that specific provable properties concerning this dissemination are satisfied.

In this chapter , we present an authenticated algorithm implementing provable reliable send primitive. This primitive is used for solving Byzantine consensus in signature-free asynchronous distributed systems.

4.2 System model and synchrony assumptions

We consider a message-passing system consisting of a finite set Π of $n(n > 1)$ processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process executes steps (send a message, receive a message or execute local computation). Value t denotes the maximum number of processes that can exhibit a Byzantine behavior. A Byzantine process may behave in an arbitrary manner. It can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, send different values to different processes, perform arbitrary state transitions, etc. A correct process is one that does not Byzantine. A faulty process is the one that is not correct. Processes communicate and synchronize with each other by sending and receiving messages over a network. The link from process p_i to process p_j is denoted $p_i \rightarrow p_j$. Every pair of process is connected by two links $p \rightarrow p_j$ and $p_j \rightarrow p_i$. Links are assumed to be reliable: they do not create, alter, duplicate or lose messages. Processes are partially synchronous, in the sense that there are unknown bounds on relative speed of a correct process . Hereafter, we define more formally a synchrony assumption required by our implementation of provable reliable send.

Definition 1. A link from a process p_i to any process p_j is timely at time τ if no message sent by p_i at time τ is received at p_j after time $(\tau + \Delta)$ or p_j is not correct.

Definition 2. A process p_i is $\langle x \rangle$-sink at time τ if: p_i is correct process and there exists a set X of correct processes (including itself) of size x, such that: for any process p_j in X, a link from p_j to p_i is timely at time τ. This means that p_i has x incoming synchronous links.

Definition 3. A process p_i is an $\diamond\langle x \rangle$-sink if there is a time τ such that, for all $\tau' \geq \tau$, p_j is an $\langle x \rangle$-sink at τ'.

Notation The notation $[\emptyset]$ (BAMP stands for Byzantine Asynchronous Message Passing) is used to denote the previous basic Byzantine asynchronous message-passing computation model. In the following, this model is both restricted with a constraint on t and enriched with additional assumption about synchrony . More precisely, $[n > 3t]$ denotes the model $[\emptyset]$ where the number of faulty processes is smaller than $n/3$, and $[n > 3t, RSA, \langle t+1 \rangle\text{-source}]$ denotes the model $[n > 3t]$ enriched with an authentication mechanism such as RSA and a synchrony assumption satisfied by a $\langle t+1 \rangle$-sink.

4.3 Provable reliable send Primitive

Provable Reliable Send is a primitive that can be used by a process p_i to send a message QUERY(m) to p_j such that a third process p_r gets a proof that QUERY(m) is in transit. Provable reliable send is defined by the following three primitives:

1. *Psend* QUERY(m, p_j): if a process p_i invokes *Psend* QUERY(m, p_j), we say that p_i *psends* QUERY(m) to p_j;
2. *Preceive* QUERY(p_i, m) : if a process p_j invokes *Preceive* QUERY(p_i, m), we say that p_j *preceives* QUERY(m) from p_i;
3. *Gproof* QUERY(m, p_i, p_j) : if a process p_r invokes *Gproof* QUERY(m, p_i, p_j), we say that p_r *getsproof* of QUERY(m) from p_i to p_j.

Formally, provable reliable send is defined by the following fourth properties:

- Integrity: A correct process p_j *preceives* QUERY(m) from a correct process p_i at most once, and only if p_i has previously *psent* QUERY(m) to p_j;
- Validity: If some correct process p_i *psends* QUERY(m) to some correct process p_j then eventually p_j *preceives* QUERY(m) from p_i;
- Proof-Integrity: If some correct process p_r *getsproof* of QUERY(m) from some process p_i to some correct process p_j, then p_j *preceives* QUERY(m) from p_i;
- Proof-Validity: If some correct process p_i *psends* QUERY(m) to some correct process p_j then every correct process p_r *getsproof* of QUERY(m) from p_i to p_j.

This primitive considers eventually timely provable reliable send, which guarantees that if the final receiver p_j of the message QUERY(m) is a bisource then eventually QUERY(m) cannot be received too much later than the proof. Formally, they define the following eventual timeliness propriety:

For a system that does not need authentication, if a process p_j is a bisource then there exists τ and T such that if some correct process p_r getsproof of QUERY(m) from some pro-

Each process p_i executes the following

(01) To *Psend* QUERY(m) to p_j;
(02) *send* QUERY(m, p_i, p_j) to all;
(03) **upon** receive QUERY(m, s, d) from p_k
(04) **if** $(s = p_k)$ and $i \neq k$ **then** *send* QUERY(m, p_k, d) to all;
(05) **if**$(p_i = d)$ and (received QUERY(m, s, d) from at least One process
and not already *Preceive* QUERY(m, s) **then** *Preceive* QUERY(m, s));
(06) **if** received QUERY(m, s, d) from $(n - t)$ distinct processes **then** *Gpoof* QUERY(m, s, d));

Figure 4.1: A Provable Reliable Send Algorithm in $[n > 3t, RSA, \langle t + 1\rangle$-source$])$

cess p_i to process p_j at time τ then p_j perceives QUERY(m) from p_i by time $max\{\tau, T\} + \Delta$.

4.4 An Algorithm Implementing Provable Reliable Send in $[n > 3t, RSA, \langle t + 1\rangle$-source$]$

Figure 4.1 presents an algorithm implementing provable reliable send primitive. It assume that an authentication mechanism such us RSA is available. A public key cryptography signatures is used by a process to verify the identity of the original sender of the message and to force a Byzantine process to relay the original message that it's received if it decides to relaying it.

If a correct process p_i invokes *Psend* QUERY(m) to p_j then p_i send a message QUERY(m, p_i, p_j) to all processes (lines 01-02). When p_i receives a message QUERY(m, s, d) from p_k, if p_k is the original sender $(s = p_k)$ and p_i is not the original sender (this is to prevent a process p_i to send a same message an infinite times), then p_i sends QUERY(m, p_k, d) to all processes (lines 03-04). If p_i is the final destinator $(d = p_i)$ of a message QUERY(m, s, d) and it receives this message from at least one process then p_i invokes *Preceive* QUERY(s, m) (line 05), if it has not previously invoked. If p_i has received QUERY(m, s, d) from $(n - t)$ distinct processes then p_i invokes *Gproof* QUERY(m, s, d) (line 06).

4.5 Conclusion

In this chapter, we presented an algorithms that implements eventually timely provable reliable send primitive in the system model$[n > 3t, RSA, \langle t + 1\rangle$-source$]$. This implementations guarantee that a message sent by a correct sender p_i will arrive with timely way to the receiver p_j if a third party p_r getsproof of this message from the original sender to the receiver and the receiver is a $\langle t + 1\rangle$-source , even the link between the sender and the receiver is asynchronous.

Chapter 5

Asynchronous Multi-valued Byzantine Consensus with Little Synchrony

5.1 Introduction

This chapter tackles the consensus problem in asynchronous systems prone to Byzantine failures. One way to circumvent the FLP impossibility result consists in adding synchrony assumptions. This chapter considers three system models, which are weaker than all previously proposed models where the Byzantine consensus can be solved in deterministic manner. The first model assumes at least one correct process connected with $2t$ privileged neighbors with eventually timely outgoing and incoming links, whereas the second assumes at least one correct process with $2t$ outgoing eventually timely links and $2t$ incoming eventually timely links. The second model is a relaxation of the former as it does not consider pair-wise links such that each pair of links connects a same pair of processes in each direction. Finally,the latter model is similar to the second, but a correct process could have $3t$, instead of $2t$, outgoing eventually timely links and $3t$,instead of $2t$ incoming eventually timely links. In those system models, three Byzantine consensus protocols are proposed. Both first protocols use authentication, but the latter one is a signature-free protocol.

5.2 Computation Model and the Consensus Problem

5.2.1 Computation Model

The system model is patterned after the partially synchronous system . The system is made up of a finite set Π of n ($n > 1$) fully-connected processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. Moreover, up to t processes can exhibit a *Byzantine* behavior, which means that such a process can behave in an arbitrary manner. This is the most severe process failure model: a Byzantine process can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, send different values to different processes, perform arbitrary state transitions, etc. A process that exhibits a Byzantine behavior is called *faulty*. Otherwise, it is *correct*.

Communication network The communication network is reliable in the sense that a message sent by a correct process to another correct process will be received exactly once within a finite time. Messages are not altered by the link and the receiver knows who the sender is. In other words, we are using authenticated asynchronous links. Such a communication network can be built atop of fair lossy links which is the classical communication model used when dealing with Byzantine processes (in fair lossy links, a message can be lost a finite number of times). They prove that even a simple retransmission until acknowledgment protocol suffices to implement a reliable link between correct processes. Using these techniques, a message that was initially lossy will eventually be received by its receiver if the sender and the receiver are correct. Note that the simulation preserves the timeliness of the messages sent on timely fair-lossy links.

Synchrony properties Every process executes an algorithm consisting of atomic computing steps (send a message, receive a message or execute local computation). We assume that processes are partially synchronous, in the sense that every correct process takes at least one step every θ steps of the fastest correct process (θ is unknown). Instead of real-time clocks, time is measured in multiples of the steps of the fastest process . In particular, the (unknown) transfer delay bound δ is such that any process can take at most δ steps while a timely message is in transit. Hence, we can use a simple step-counting for timing out messages. Hereafter, we define more formally a timely link, an x-bisource and an x-SD.

Definition 4. A link from a process p_i to any process p_j is timely at time τ if (1) no message sent by p_i at time τ is received at p_j after time $(\tau + \delta)$ or (2) process p_j is not correct.

Definition 5. A process p_i is an x-bisource at time τ if:
- (1) p_i is correct
- (2) There exists a set X of processes of size x, such that: for any process p_j in X, both links from p_i to p_j and from p_j to p_i are timely at time τ. The processes of X are said to be privileged neighbors of p_i.

Definition 6. A process p_i is an $\diamond x$-bisource if there is a time τ such that, for all $\tau' \geq \tau$, p_i is an x-bisource at τ'.

Definition 7. A process p_i is an x-SD (S for Source and D for destination) at time τ if:
- (1) p_i is correct
- (2) There exists two sets X and Y of processes of size x such that for any process p_j in X the link from p_i to p_j is timely at time τ and for any process p_k in Y the link from p_k to p_i is timely at time τ. If $X = Y$ then p_i is an x-bisource. The processes of X are said to be privileged out-neighbors of p_i and the processes of Y are said to be privileged in-neighbors of p_i.

Definition 8. A process p_i is an $\diamond x$-SD if there is a time τ such that, for all $\tau' \geq \tau$, p_i is an x-SD at τ'.

5.2.2 Byzantine behavior and authentication

A Byzantine process is a process the behavior of which can deviate from its specification. Such a process can send information that does not comply with the text of the protocol it is intended to execute (sending more messages than expected, sending messages with wrong headers, sending values from a different type than expected, etc.). This behavior can be easily detected and the incriminated process is tagged Byzantine and is ignored (all his messages are ignored and no information is no more expected from it). A second behavior is that a Byzantine process can remain silent which corresponds to a crash failure. A Byzantine process can also send different values to different processes. For example, let p_b be a Byzantine process which is expected to send a value from the set $\{x, y\}$ to some processes. At some point of the execution, process p_b sends a value x where it should send value y. A receiving process cannot know that p_b is Byzantine as the received value is a plausible value. Finally, a Byzantine process can send a value x to some processes and a value y to the others. If the message is properly formed (according to the protocol), the receiving processes can exhibit inconsistent behaviors as they receive inconsistent data. This behavior is called duplicity. Finally, a Byzantine process can send a wrong, but plausible value.

In order to deal with these behaviors, the proposed protocols will use some mechanisms. Let us first give an idea of the programming model of the protocols we propose. The proposed protocols are composed of a series of communication steps. During a step, each of a given number of processes (a priori known) send one message to a given set of processes (also known a priori). Then each process waits for a received message until some predicate becomes true (a timer times-out or a minimum number of messages is received). Finally, according to the received messages, some local computation is done.

During a communication step, a process p_i may have to relay a value v it has received from a process p_j during the previous step. In order to prevent p_i from sending a value $w (w \neq v)$. The proposed protocols use application level signatures (public key cryptography such as RSA signatures). This means that the value received by p_i from p_j at the previous step was signed by p_j. p_i can read and use the value by deciphering it but in order to relay it, p_i has to forward the signed value. By this means, p_i cannot relay w if we assume that p_i cannot forge the signature of p_j. If p_i forwards any value different from v, it will be detected and the receiver will know that p_i is Byzantine. The only bad behaviors p_i can exhibit in this case without being discovered are (1) to remain silent, (2) to send a default value (e.g. $\perp$) meaning that it received no value from p_j or (3) to send the right value v. Consequently, cryptography allows us to reduce the power of Byzantine processes. Of course, this means that in our model we assume that Byzantine processes are not able to subvert the cryptographic primitives.

Let us consider a second scenario. During step s, each process sends a value to all other processes. Then, each process waits for as many messages as possible. As at most t processes may exhibit a Byzantine behavior, a process can face the situation where all of the t Byzantine processes decide to remain silent. In order not to block forever waiting for messages, a given process cannot expect more than $n - t$ message during a general exchange of messages. When the waiting predicate holds, each process keeps the maximal value it received. This value will be sent to all processes during the next step $s + 1$. Let us consider the case where all processes send at step s the value 0 except p_n that sends the value 1. To prevent a Byzantine process p_b to send a value v different from 0 and

1, each message has to carry a value and the set of $n - t$ values received by p_b during the previous step s. The included signed values can be used by a receiving process to check whether the value sent by process p_b complies with the values p_b received at step s. This set of signed values is called *certificate* and its role is to prove to the receiver that the value is legal. Indeed, as process p_b can receive only values 0 and at most one value 1, no set of signed values can justify a value v different from 0 and 1. Consequently p_b is obliged to send only possible values (of course, p_b can remain silent and thus sends nothing). However, if process p_b received all of the n values of step s, it can build two sets of values one that includes the values of processes ranging from 1 to $n - t$ and a second set of values that includes the values of processes ranging from $t + 1$ to n. The two sets can justify respectively the values 0 and 1. Consequently, p_b can send value 0 to some processes and value 1 to the others without being discovered as both values are possible. This means that the use of certificates does not prevent Byzantine process from sending different possible values to different processes.

Finally, in order not to add to the protocol code that is not directly related to the proposed solution, we assume that each process has an underlying daemon that filters the messages it receives. For example, the daemon will discard all duplicate messages (necessarily sent by Byzantine processes as we assume reliable send and receive operations between correct processes). The daemon, will also discard all messages that are not syntactically correct, or that do not comply with the text of the protocol (e.g. a process that sends two different messages with the same type within the same communication step, a process that sends a message to a wrong process, etc.). Of course a message that do not comply with the associated certified is also discarded.

5.2.3 The Consensus Problem

The Consensus problem has been informally stated in the Introduction. This paper considers *multivalued* Consensus (no bound on the cardinality of the set of proposable values): every process p_i *proposes* a value v_i and all correct processes have to eventually *decide* on some value v in relation with the set of proposed values. Let us observe that, in a Byzantine failure context, the consensus definition should not be too strong. For example, it is not possible to force a faulty process to decide the same value as the correct processes, since a Byzantine process can decide whatever it wants. Similarly, it is not reasonable to decide any proposed value since a faulty process can initially propose different values to distinct processes and consequently the notion of "proposed value" may not be defined for Byzantine processes. Thus, in such a context, the consensus problem is defined by the following three properties:

- Termination: Every correct process eventually decides.
- Agreement: No two correct processes decide different values.
- Validity: If all the correct processes propose the same value v, then only the value v can be decided.

5.3 An Authenticated Byzantine Protocol With $\diamond 2t$-bisource

The protocol of (Figure 5.1) uses authentication and assumes an $\diamond 2t$-bisource. Each process p_i manages a local variable est_i which contains its current estimate of the decision value. The init phase (lines 01-03) consists of an all-to-all message exchange that allows to initializes the variable est_i to a value that is received at least $(n - 2t)$ times if any[1]. Otherwise, est_i is set v_i the value proposed by p_i. This phase establishes the validity property. Indeed, if all correct processes propose the same value v, all processes will receive v at least $(n - 2t)$ times and the only value that can be received at least $(n - 2t)$ times is v. In such situation, when the different processes will proceed to the next rounds v will be the only certified value (its certificate includes the $(n - 2t)$ values v signed by their senders and received during the init phase). From line 05 of the protocol, when a process sends a value x it signs, it associates with it a certificate composed of the signed values received during the previous phase and that led the process to keep this value.

After the init phase, the protocol proceeds in consecutive asynchronous rounds. Each process p_i manages a variable r_i (initially set to 0). Each round r is coordinated by a predetermined process p_c (*e.g.*, c can be defined according to the round robin order). So, the protocol uses the well-known *rotating coordinator* paradigm. Each round is composed of four communication phases.

First phase of a round r (lines 05-07). Each process that starts a round (including its coordinator) first sends its own estimate (with the associated certificate) to the coordinator (p_c) of the current round and sets a timer to ($\Delta_i[c]$). Δ_i is an array of time-outs (one per process) managed by p_i. Each entry $\Delta_i[j]$ keeps the estimation of p_i of the round trip delay from p_i to p_j followed by a local computation on p_j and message return from p_j to p_i. This value is set to 1 and each time p_i sends a time-constrained message to p_j, a timer is set to $\Delta_i[j]$. If the timer times out while waiting for the response from p_j, $\Delta_i[j]$ is incremented and p_i considers that either p_j is not a privileged neighbor or p_j is Byzantine or the value $\Delta_i[j]$ is not set to the right value. As $\Delta_i[j]$ is incremented each time p_j's responses misses the deadline, it will eventually reach the bound on the round trip between p_i and p_j if p_i and p_j are privileged neighbors. Moreover, this prevents p_i from blocking while waiting (line 06) for the response of a faulty coordinator.

For any round, the coordinator will receive at least $(n-t)$ QUERY messages but it will send COORD messages only once and will ignore and subsequent QUERY messages related to the same round. When the coordinator of round r receives at line 19 a valid QUERY message (perhaps from itself) containing an estimate est for the first time, it sends a COORD(r, est) message to all processes. The COORD message is sent from another parallel task because the coordinator of round r could be waiting for other messages in previous rounds and if it does not responds quickly, the sender of the QUERY message may time out. This is why, whatever is the coordinator doing, as soon as it receives a valid QUERY message for a round it coordinates, it sends the included estimate to all processes (this allows a coordinator to coordinate a round with a certified value it has received even if it is itself lying far behind). when a process p_i receives the value of the coordinator, it stores it in

[1]This phase does not use certificates as there is no prior communication (the exchanged values are not signed).

its variable aux_i otherwise this variable is set to a default value $\perp$ meaning that no value is received from the coordinator of this round.

If the current coordinator is a $2t$-bisource and the round-trip delays (Δ array) with its $2t$ privileged neighbors are correctly set then at least $(t+1)$ correct processes will get the value v of the coordinator and thus set their variable aux to v ($\neq \perp$). The next phases will serve to propagate this value from the $(t+1)$ correct processes to all correct processes. Indeed, among the $2t$ privileged neighbors of the current coordinator at least t are correct processes and all of them will receive the value of the coordinator before their timers time-out. If we add the coordinator itself we have $(t+1)$ correct processes. If the current coordinator is Byzantine, it can send nothing to some processes and/or perhaps send different certified values to different processes (in such a case, necessarily none of these values has been decided in a previous round as we will see later). If the current coordinator is not a $2t$-bisource or if it is Byzantine, the three next phases allow correct processes to behave in a consistent way. Either none of them decides or if some of them decides a value v despite the Byzantine behavior of the coordinator, then the only certified value for the next round will be v preventing Byzantine processes from introducing other values.

Second phase of a round r (lines 08-10). This phase aims to extend the scope of the $2t$-bisource. Indeed, if the current coordinator is a $2t$-bisource then at least $(t+1)$ correct processes set their variable aux_i to the same non-$\perp$ value (say v). During the second phase, all processes relay, using an all-to-all message exchange, the value they got from the coordinator (with its certificate) or $\perp$ if they timed out. Each process p_i collects $(n-t)$ valid messages (the deciphered values carried by these messages are stored into a set V_i - of course each value appears at most once in V_i as V_i is a set). If the coordinator is a $2t$-bisource then any correct process will get at least one message from the set of $(t+1)$ correct processes that got the value of the coordinator because $(n-t)+(t+1) > n$. If the coordinator is not a $2t$-bisource or if it is Byzantine, some processes can receive only $\perp$ values, others may receive more than one value (the coordinator is necessarily Byzantine in this case) and some others can receive a unique value. This phase has no particular effect in such a case. The condition $(V_i - \{\perp\} = \{v\})$ of line 10 means that if there is only one non-$\perp$ value v in V_i then this value is kept in aux_i otherwise, aux_i is set to $\perp$.

Third phase of a round r (lines 11-13). This phase has no particular effect if the coordinator is correct. Its aims is to deal with the situations where the coordinator is Byzantine. Indeed, in such a case two different correct processes may have set their aux_i variables to different values during the second phase. Phase three is thus a filter, it ensures that at the end of this phase, at most one non_$\perp$ value is kept in the aux variables. In other words, at the end of this phase, if p_i and p_j are two correct processes and if $aux_i \neq \perp$ and $aux_j \neq \perp$ then necessarily $aux_i = aux_j$ whatever is the behavior of the Byzantine processes. This phase consists of an all-to-all message exchange. Each process collects $(n-t)$ valid messages the values of which are stored in a set V_i. If all received messages contain the same value v ($V_i = \{v\}$) then v is kept in aux_i otherwise aux_i is set to the default value $\perp$. At the end of this phase, there is at most one (or none) certified value v ($\neq \perp$).

Fourth phase of a round r (lines 14-17). This phase is the decision phase. Processes try

to decide. For a process to decide a value v, it has to ensure first that this value will be the only certified value for next rounds and will be known by all correct processes (such a value is then said to be locked). For this, processes exchange their aux_i variables using an all-to-all message exchange. Recall that after phase three, there is at most one certified value (say v) in aux_i variables. This means that at the beginning of the fourth phase a process (whether correct or Byzantine) can send only the unique value v (if any) or $\perp$ (otherwise the message will be non valid). Processes collect $(n-t)$ valid messages and store the values in V_i. If the set V_i of a process p_i contains a unique non_$\perp$ value v, p_i decides v. Indeed among the $(n-t)$ same values v received by p_i, at least $n-2t$ have been sent by correct processes. As $(n-t)+(n-2t) > n$ any set of $(n-t)$ valid signed messages of this phase includes at least one value v. Hence, all processes receive at least one value v (the other values could be v or $\perp$) and the only certified value for the next rounds is v. This means that during the next round (if any) no coordinator (whether correct or Byzantine) can send a valid value different from v.

If during the fourth phase, a process p_i receives only $\perp$ values, it is sure that no process can decide during this round and thus it can keep the value it has already stored in est_i (the certificate composed of the $(n-t)$ valid signed messages received during phase four containing $\perp$ values, allow p_i to keep its previous values est_i).

Before deciding (line 16), a process first sends to all other processes a signed message DEC that contains the decision value (and the associated certificate). This will prevent the processes that progress to the next round from blocking because some correct processes have already decided and stopped sending messages. When a process p_i receives a valid DEC message at line 20, it first relays is to all other processes and then decides. Indeed, task T_3 is used to implement a reliable broadcast to disseminate the eventual decision value preventing some correct processes from blocking while others decide (not all processes decide necessarily during the same round).

5.4 An Authenticated Byzantine Consensus Protocol with a $\Diamond 2t$-SD

The protocol of (Figure 5.2) is an extension of the first protocol. The protocol uses authentication and assumes an $\Diamond 2t$-SD ($2t$ outgoing eventually synchronous links and $2t$ incoming eventually synchronous links). The principle of protocol is similar to the protocol that assumes an $\Diamond 2t$-bisource except for the coordination phase of each round (lines 05-07) that are replaced by lines 101-104. Each process manages same local variables as in the first protocol. When a given process p_i (including the coordinator of the round) starts a round r it first sends its own estimate (with the associated certificate) to all the processes (in the previous protocol, the message is sent only to the coordinator of the round). Moreover, instead of arming the timer just after this sending, each process has to wait the reception of at least $(n-t)$ QUERY$(r, *)$ messages before arming the timer. the protocol then behaves like the previous one. If a COORD(r, est) is received from the coordinator of the current round, the timer is disabled and if the timer times-out, the aux_i variable is set to the default value $\perp$ and the estimation of the response time $\Delta_i[c]$ is incremented.

The intuition that is behind this modification is the following. When we assume an $\Diamond 2t$-bisource, there is a time after which a privileged neighbor p_i is sure that for any request

Function Consensus(v_i)

Init: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;

Task $T1$: % basic task %

——————————————————————- init phase ——————————————————————

(01) *send* INIT(v_i) to all;
(02) **wait until** (INIT messages received from at least $(n-t)$ distinct processes);
(03) **if** ($\exists v$: received at least $(n-2t)$ times) **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$ **endif**;

repeat forever
(04) $c \leftarrow (r_i \mod n) + 1$; $r_i \leftarrow r_i + 1$;

——————————————————————- round r_i ——————————————————————

(05) *send* QUERY(r_i, est_i) to p_c; *set_timer*($\Delta_i[c]$);
(06) **wait until** (COORD(r_i, est) received from p_c or *time-out*) **store value** in aux_i; % *else* $\perp$ %
(07) **if** (*timer times out*)) **then** $\Delta_i[c] \leftarrow \Delta_i[c] + 1$ **else** *disable_timer* **endif**;

(08) *send* RELAY(r_i, aux_i) to all;
(09) **wait until** (RELAY($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(10) **if** ($V_i - \{\perp\} = \{v\}$) **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(11) *send* FILT1(r_i, aux_i) to all;
(12) **wait until** (FILT1($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(13) **if** ($V_i = \{v\}$) **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(14) *send* FILT2(r_i, aux_i) to all;
(15) **wait until** (FILT2($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(16) **case** ($V_i = \{v\}$) **then** *send* DEC(v) to all; **return**(v);
(17) ($V_i = \{v, \perp\}$) **then** $est_i \leftarrow v$;
(18) **endcase**;

———

end repeat

Task $T2$: % coordination task %
(19) **upon** *receipt* of QUERY(r, est) for the first time for round r: *send* COORD(r, est) to all;

Task $T3$:
(20) **upon** *receipt* of DEC(est): *send* DEC(est) to all; **return**(est);

Figure 5.1: An Authenticated Byzantine Consensus Protocol (assumes a $\diamond 2t$-bisource)

(101) *send* QUERY(r_i, est_i) to all;
(102) **wait until** (QUERY$(r_i, *)$ received from at least $(n-t)$ distinct processes); *set_timer*$(\Delta_i[c])$;
(103) **wait until** (COORD(r_i, est) received from p_c or *time-out*) **store value** in aux_i; % *else* $\perp$ %
(104) **if** (*timer times out*)) **then** $\Delta_i[c] \leftarrow \Delta_i[c] + 1$ **else** *disable_timer* **endif**;

Figure 5.2: An Authenticated Byzantine Consensus Protocol (assumes a $\diamond 2t$-SD)

sent to the bisource it will receive the response before the time-out. This is due to the fact that by the model assumption, the two links is both direction between p_i and the bisource are timely and the processes are eventually synchronous. This means that p_i can no more miss the COORD(r, est) messages of the bisource. In the present situation, a privileged neighbor of the SD is connected to it either by an outgoing link or an incoming link that is eventually timely. This is why lines 05-07 of the first protocol need to be changed. Let p_i be a privileged neighbor of the $\Diamond 2t$-SD with which it is connected by an eventually timely outgoing link. Similarly to the previous case, there is a time after which all the links that compose the SD are timely. This means that when process p_i receives the $(n-t)$ messages at line 102 during a round coordinated by the SD, it is sure that at least $(n-t)$ processes sent a QUERY$(r, *)$ message to the SD. As the SD has $2t+1$ timely incoming links ($2t$ of the $\Diamond 2t$-SD and the SD itself) among which at least $t+1$ are correct processes, as there are at most t Byzantine processes in the system, p_i is sure that at least one message QUERY$(r, *)$ is already sent by some process p_j to the SD on a timely link. This why p_i can set a timer and wait for a time delay that corresponds to a transfer delay from p_j to the SD plus the local computation time on the SD and then a transfer delay from the SD to p_i. By the model assumption, this delay is eventually bounded by some value b and as the local variables Δ_i are incremented after every time-out, there will be a time after which the $\Delta_i[]$ delay will reach the value b. Thus process p_i that is connected with an eventually timely incoming link with the SD will never time-out while waiting the COORD(r, est) messages of the SD.

5.5 A Byzantine Consensus Protocol In Signature-Free Systems with a $\Diamond 3t$-SD

This section presents a protocol (Figure 5.5) that solving the Multivalued Byzantine Consensus problem in a signature-free system (without message authentication). This protocol assumes an $\diamond 3t$-SD($3t$ outgoing eventually synchronous links and $3t$ incoming eventually synchronous links) and it uses as primitive Reliable-Broadcast, which is very similar to consistent unique broadcast and consistent broadcast . Note that $\diamond 3t$-SD is equivalent to $\diamond 3t$-bisource in the case one the number of processes in the system is exactly $(n = 3t + 1)$.

5.5.1 A Simple Reliable-Broadcast Algorithm

Figure 5.3 presents a simple algorithm that implementing reliable broadcast in an asynchronous Byzantine system where $t < n/3$. This algorithm uses three message types (INIT(), ECHO() and READY()) and two operation R_broadcast() and R_deliver(). When a process p_i invokes R_broadcast(v_i), it broadcasts the message INIT(v_i, i). When a process

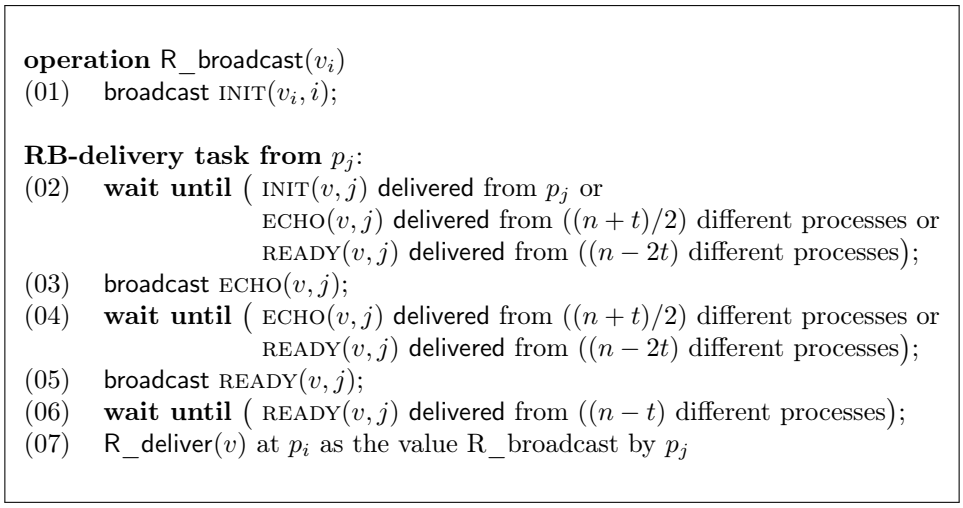

Figure 5.3: A Simple Reliable-Broadcast Algorithm

p_i has delivered a message INIT(v, j) from p_j or from ECHO(v, j) from $((n+t)/2)$ distinct processes READY(v, j) from $((n-2t)$ (Lines 01, 02), ECHO(v, j) (line 03). When p_i has delivered a message ECHO(v, j) delivered from $((n+t)/2)$ distinct processes or READY(v, j) from $((n-2t)$ distinct processes (Line 04),it broadcasts READY(v, j) (Line 05). then p_i waits until it has delivered the same message READY(v, j) from $(n-t)$ distinct processes. When this occurs it R_delivers value (v) as the value R_broadcast by p_j (Lines 06, 07). Reliable-broadcast ensures the following properties:

- No-duplicity property: no two correct processes R_deliver different messages from any p_j
- Termination property: if the sender is correct, all correct processes eventually R_deliver its message.
- Uniformity property: if a correct process R_delivers a message from p_j (possibly faulty), then all correct R_deliver a message from p_j

5.5.2 An extension of Reliable-Broadcast to get a weaker delivery

The algorithm of figure 5.4 presents an extension of Reliable-Broadcast presented in figure 5.3.The principle of this algorithm is similar to the precedent one except for the lines 06-07 that are replaced by lines 601-602. The intuition that is behind this modification to get Weaker delivery propriety, denoted WR_deliver. When a process p_i has delivered the message READY(v, j) from $(t+1)$ distinct processes, it WR_delivers value v as the value R_broadcasted by p_j. This means that p_i receives a message READY(v, j) from at a least one correct process. The Reliable-broadcast with weaker delivery guarantees the following properties:

- No-duplicity property: If a correct process p_i WR_delivers a value v as the value R_broadcasted by p_k (possibly faulty) ,then a correct processp_j will never WR_deliver the message v of p_k, p_j WR_deliver the same value v or p_j R_deliver same value v.

(601) **wait until** (READY(v, j) delivered from $(t+1)$ different processes);
(602) WR_deliver(v) at p_i as the value R_broadcast by p_j

Figure 5.4: An extension to reliable-broadcast to get a weaker delivery

- Termination property: if the sender is correct, all correct processes eventually WR_deliver its message.
- Weak Uniformity property: if a correct process R_delivers a message from p_j (possibly faulty), then all correct WR_deliver a message from p_j

We also define a timeliness propriety, to get eventually timely weaker delivery , as the following:

- If p_i is $\diamond 3t$-SD, then If a correct processes p_j R_delivers a value v as the value R_broadcasted by p_k (possibly faulty), then p_i will WR_deliver certainly the same value v of p_k in a bounded time.

5.5.3 Description of the proposed protocol

Figure 5.5 presents a protocol that solving the Multivalued Byzantine Consensus problem in an asynchronous distributed system. The proposed protocol is a signature-free (does not use authentication) and assumes an $\diamond 3t$-SD, it uses as subroutines Reliable-Broadcast 5.3 and Reliable-Broadcast with timely weaker delivery 5.3. Those subroutine abstracts three operations: R_broadcast(). R_deliver() and WR_deliver().

Each process in the system executes consecutive asynchronous rounds and each round r is coordinated by a predetermined process p_c. Each round is composed by five phases.

First phase of a round r(lines 01-06). This phase aims to certify values

carried by messages. Each process p_i manages a local variable est_i which contains its current estimate of the decision value. In this phase, a process p_i uses Reliable-Broadcast operations (R_broadcast() and R_deliver()) to R_brodacst CERT(r_i, est_i) message. If p_i R_delivers CERT(r_i, est) message from at a least $(n-t)$ distinct processes then it stores values carried by those message in a set $rec1_i$. A process $p-i$ considers a value v to be certified if it R_delivers v from at a least $(n-2t)$ distinct processes (at a least one is correct). A processes p_i considers also the value $(\perp)$ as certified if the exist a set $A \subset rec1_i$ with $|A| \geq (n-t)$ and $\nexists v \in A$ such that $\#_v(rec1_i) \geq n-2t)\}$. This means that p_i certifies the value $\perp$ if no value v appeared at least $(n-2t)$ times in $rec1_i$. a process p_i stores the certified Values in a set $valid1(rec1_i)$ and sets its variable aux_i to any value of $valid1(rec1_i)$.

Second phase of a round r (lines 07-12). This phase guarantees that at its end, at most one non $\perp$ value can be kept in the aux variables. In other words, if p_i and p_j are correct processes and if $aux_i \neq \perp$ and $aux_j \neq \perp$ then necessarily $aux_i = aux_j$, whatever is the

behavior of the Byzantine processes.

In this phase, each process p_i R_broadcasts $R\_broadcast$ FILT(r_i, aux_i) message and waits to R_deliver FILT(r_i, aux),with $(aux \in valid1(rec1 - i))$ from at least $(n - t)$ distinct processes

When p_i _delivers these messages, the values of which are stored in a set $rec2_i$. A process p_i considers v as a valid value, If $rec2_i$ contains a least $(n - t)$ values for v. If no value v appeared at least $(n - t)$ times in $rec2_i$, a process p_i considers $\perp$ as valid value. Process p_i stores the valid Values in a set $valid2(rec2 - i)$. At the end of this phase, p_i updates its aux_i variable to any value of $valid2(rec2_i)$ and starts its timer on the coordinator.

Third phase of a round r (lines 13-23). This phase is the decision phase. In this phase, a process p_i R_broadcasts DEC(r_i, aux_i).When p_i R_delivers DEC(r_i, aux) message with $(aux \in valid2(rec2_i))$ from at a least $(n - t)$ then it stores the values carried by these messages in a set $rec3_i$. and updates its aux_i variable to any value of $valid2(rec2_i)$. If the set $rec3_i$ contains at least $(n - t)$ values for $v \neq \perp$ then p_i decides v and sets its est_i variable to v. If $rec3_i$ s contains at a least one value $v \neq \perp$ then p_i change its estimation to v.Otherwise, p_i the help of the coordinator is needed. This phase guarantees that the Agreement property will never be violated, because if any correct process decides v during the current round then if some processes do not decide in the same round, then v is the only certified value for the next round. Note that not all processes decide necessarily during the same round.

Forth phase of a round r (lines 24-30). This phase is the coordination phase. This phase has no particular effect if the set $rec3_i$ of a correct process p_i, contains more than one non $\perp$ value v. This means that p_i has decided on v in the previous phase or its set $rec3_i$ contains at a least one value $v \neq \perp$. Its aim is to help processes that having theirs sets $rec3$ containing less than one non $\perp$ value v to change their estimation to the coordinator value if they receiving it . In this phase, the coordinator p_c uses WR_deliver() operation defined in the algorithm of figure 5.4. The coordinator p_c waits to WR_deliver FILT(r_i, aux) message, that are R_broadcasted by all processes at a line 07 ,with $(aux \in valid1(rec1_i)$ from at least $(n - t)$ distinct processes. When it WR_delivers the values carried by these messages, then it stores them in a set rec_i Note that, all correct processes start their timers at a line 12 when each of them R_delivers at a least $(n - t)$ values carried by FILT(r_i, aux) messages. This means that, these correct processes have a proof that the coordinator will WR_deliver all values of the set $rec2_i$.

If p_c is a $\diamond 3t$-SD then it WR_delivers all value of the set $rec2_i$ with timely way. This means that all values off rec_i are WR_delivred with timely way. Moreover, if the set rec_i contains a value $v \neq \perp$ more than $(t + 1)$ times then it keeps v in its local variable $coord_i$, else it sets $coord_i$ to its own estimation. After this, p_c broadcasts COORD$(r_i, coord_i)$ message. The COORD$(r_i, coord_i)$ message is broadcasted from another parallel task, because the coordinator of round r could be stuck in previous rounds and if it does not respond quickly, the processes waiting this message may time out. When the timer a correct process p_i times out while waiting the response from p_c , $\Delta_i[j]$ is incremented. Moreover, this prevents p_i from blocking while waiting for the response of a faulty coordinator. If the current coordinator is a $\diamond 3t$-SD then it has at least $3t$ processes with which is connected

by outgoing eventually timely links among which at least $2t$ are correct processes. Consequently, at least $(2t + 1)$ correct processes (the $2t$ correct privileged in-neighbors and the coordinator itself) got the value v of the coordinator, carried by COORD$(r_i, coord)$ message, and thus set their variable $coord_i$ to the value of the coordinator $coord$. The processes that not receive the COORD$(r_i, coord)$ message set their variable $coord_i$ to $\perp$.

Fifth phase of a round r (lines 24-30). This phase is the relaying phase. This phase has no particular effect if the coordinator is not a $\diamond 3t$-SD or is a Byzantine. Certainly, if the current coordinator is a $\diamond 3t$-SD then at least $(2t + 1)$ correct processes set their variable $coord_i$ to the same value $v \neq \perp$. During the fifth phase, all processes broadcast RELAY$(r_i, coord_i)$ message . This means that they relay the value v they got from the coordinator or $\perp$ if they timed out. Each process p_i delivers RELAY$(r_i, coord_i)$ message from at a least $(n - t)$ distinct processes (the values carried by these messages are stored into a set $rec4_i$). If the coordinator is a $\diamond 3t$-SD then any correct process will get $(t + 1)$ message (at least one from a correct processes) from the set of $(2t + 1)$ correct processes that got the value of the coordinator because $(n - t) + (2t + 1) > n$. If the coordinator is not a $\diamond 3t$-SD or if it is Byzantine, some processes can receive only $\perp$ values, others may receive more than one value and some others can receive a unique value.

At the end of this phase, a process p_i that asking for the help of the coordinator to a value $v \neq \perp$, if v appears at a least $t + 1$ times in the set $rec4_i$.

5.6 Conclusion

This chapter has presented Three protocols for solving Consensus in distributed systems prone to Byzantine failures. First and second protocols use authentication and assume a relaxed partially synchronous distributed system but where only $4t$ communication links are eventually synchronous.These links connect the same process ($2t$ incoming links and $2t$ outgoing links).Those protocols have very simple design principles. In favorable setting, they can reach decision in only 6 communication steps and needs only $\Omega(n^2)$ messages in each step.

The third protocol is a signature-free protocols and assumes only $6t$ communication links are eventually synchronous. These links connect the same process ($3t$ incoming links and $3t$ outgoing links). In favorable setting, the proposed protocol can reach decision in 11 communication steps and needs $\Omega(n^3)$ messages in each step.

```
Consensus(v_i)

Init: r_i ← 0; Δ_i ← 1; est_i ← v_i

Task T_Basic:      % basic task %
     repeat forever
(01) c ← (r_i  mod n) + 1; r_i ← r_i + 1;
     ------------------------------- round r_i -------------------------------
(02) R_broadcast CERT(r_i, est_i);
(03) let rec1_i = multiset of values R_delivered to p_i and carried by CERT messages;
(04) wait until (|rec1_i| ≥ n − t);
(05) let valid1(X) ≡ {x|#_x(X) ≥ n − 2t} ∪ {⊥, if ∃A ⊂ X, |A| ≥ (n − t), ∄v ∈ A|#_v(X) ≥ n − 2t)};
(06) aux_i ← any value v ∈ valid1(rec1_i);

(07) R_broadcast FILT(r_i, aux_i);
(08) let rec2_i = multiset of values x R_delivered to p_i and carried by FILT messages with x ∈ valid1(rec1_i);
(09) wait until (|rec2_i| ≥ n − t);
(10) let valid2(X) ≡ {x|(#_x(X) ≥ n − t)} ∪ {⊥, if ∃A ⊂ X, |A| ≥ (n − t), ∄v ∈ A|#_v(X) ≥ n − t)};
(11) aux_i ← any value v ∈ valid2(rec2_i);
(12) set_timer(Δ_i[c]);

(13) R_broadcast DEC(r_i, aux_i);
(14) let rec3_i = multiset of values x R_delivered to p_i carried by DEC messages with x ∈ valid2(rec2_i);
(15) wait until (|rec3_i| ≥ n − t);
(16) aux_i ← any value v ∈ valid2(rec2_i);
(17) case (∃v ≠ ⊥, #_v(rec3_i) ≥ n − t) then decide(v); est_i ← v;
(18)      (∃v ≠ ⊥, #_v(rec3_i) > 0)      then est_i ← v;
(19)      otherwise      % The help of the coordinator is needed %
(20)                let rec4_i = multiset of values delivered to p_i and carried by RELAY messages;
(21)                wait until (|rec4_i| ≥ n − t);
(22)                if (∃v ≠ ⊥, #_v(rec4_i) ≥ t + 1)) then est_i ← v endif;
(23) endcase;
     ------------------------------------------------------------------------
     end repeat

Task T_Coord[r]:      % coordination task of round r %
(24) let rec_i = multiset of values x WR_delivered to p_i carried by FILT messages with x ∈ valid1(rec1_i);
(25) wait until (|rec_i| ≥ n − t);
(26) if (∃v ≠ ⊥, #_v(rec_i) ≥ t + 1) then coord_i ← v else coord_i ← est_i endif;
(27) broadcast COORD(r_i, coord_i);

Task T_Relay[r]:      % relay the value of the coordinator of round r %
(28) wait until (COORD(r_i, coord) delivered from p_c or time-out)
(29) if (timer times out) then coord_i ← ⊥; Δ_i[c] ← Δ_i[c] + 1 else coord_i ← coord; disable_timer; endif;
(30) broadcast RELAY(r_i, coord_i);
```

Figure 5.5: A Byzantine Consensus Protocol In Signature-Fee systems (assumes a $\diamond 3t$-SD)

Chapter 6

Time-Free Authenticated Byzantine Consensus

6.1 Introduction

This chapter presents a time-free deterministic solution to the Byzantine consensus problem. Moreover, we can see a difference with the timer-based approach. The time-free approach in the authenticated Byzantine model needs twice more winning links compared to the crash failures model whereas in the case of the timer-based approach we need four times more timely links to tolerate t Byzantine faults compared to the t links needed for crash failures. This can be explained by the query-response mechanism used by the time-free approach.

6.2 Basic Computation Model and Consensus Problem

6.2.1 Asynchronous Distributed System with Byzantine Process

We consider a message-passing system consisting of a finite set Π of n ($n > 1$) processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process executes steps (send a message, receive a message or execute local computation). Value t denotes the maximum number of processes that can exhibit a Byzantine behavior. A Byzantine process may behave in an arbitrary manner. It can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, send different values to different processes, perform arbitrary state transitions, etc. A correct process is one that does not Byzantine. A faulty process is the one that is not correct.

Processes communicate and synchronize with each other by sending and receiving messages over a network. The link from process p to process q is denoted $p \rightarrow q$. Every pair of process is connected by two links $p \rightarrow q$ and $q \rightarrow p$. Links are assumed to be reliable: they do not create, alter, duplicate or lose messages. There is no assumption about the relative speed of processes or message transfer delays. We assume that an authentication mechanism along with a public key infrastructure and a public key cryptography such as RSA signatures are available. We assume that Byzantine processes cannot imperson-

ate other processes. Moreover, processes sign the messages they send. Consequently, a Byzantine process cannot alter or modify a message it relays as it cannot forge the signature of the original sending process. In our authenticated Byzantine model, we assume that Byzantine processes are not able to subvert the cryptographic primitives. To ensure the message validity, each process has an underlying daemon that filters the messages it receives. For example, the daemon will discard all duplicate messages (necessarily sent by Byzantine processes as we assume reliable send and receive operations between correct processes). The daemon, will also discard all messages that are not syntactically correct, or that do not comply with the text of the protocol.

6.2.2 A Time-Free Assumption

Query-Response Mechanism In this chapter, we consider that each process is provided with a query-response mechanism. More specifically, any process p can broadcast a QUERY () message and then wait for corresponding RESPONSE () messages from $(n-t)$ processes. Each of this RESPONSE () messages is a winning response for that query, and the corresponding sender processes are the winning processes for that query. The others responses received after the $(n-t)$ RESPONSE () messages are the losing responses for that query, and automatically discarded. A process issues a new query only when the previous one has terminated (the first $(n-t)$ responses received). Finally, the response from a process to its own queries is assumed to always arrive among the first $(n-t)$ responses that is waiting for.

Henceforth, we define formally a winning link, an x- winning.

Definition 9. Let p and q be two processes. The link $p \to q$ is eventually winning (denoted $\diamond WL$) if there is a time τ such that the response from p to each query issued by q after τ is a winning response (τ is finite but unknown).

Definition 10. A process p is an x-winning at time τ if p is correct and there exists a set X of processes of size x, such that: for any process q in X, the link $p \to q$ is winning. The processes of X are said to be privileged neighbors of p.

Definition 11. A process p is an $\diamond x$-winning if there is a time τ such that, for all $\tau' \geq \tau$, p is an x-winning at τ'.

For the rest of the chapter, we consider an asynchronous distributed system where the only additional assumptions are those needed by the $\diamond x$-winning.

6.2.3 The Consensus Problem

We consider the multivalued consensus problem, where there is no bound on the cardinality of the set of proposable values. In the multivalued consensus problem, every process p_i proposes a value v and all correct processes have to eventually decide on a single value among the values proposed by the processes.

Formally, the consensus problem is defined by the following three properties:

Let us observe that, in a Byzantine failure context, the consensus definition should not be too strong. For example, it is not possible to force a faulty process to decide the

same value as the correct processes, since a Byzantine process can decide whatever it wants. Similarly, it is not reasonable to decide any proposed value since a faulty process can initially propose different values to distinct processes and consequently the notion of "proposed value" may not be defined for Byzantine processes. Thus, in such a context, the consensus problem is defined by the following three properties:

- Termination: Every correct process eventually decides.
- Agreement: No two correct processes decide different values.
- Validity: If all the correct processes propose the same value v, then only the value v can be decided.

6.3 An Authenticated Byzantine Consensus Protocol With $\diamond 2t$-winning

The proposed protocol (Figure 6.1) uses authentication and assumes an $\diamond 2t$-winning process. Except the coordination phase at the beginning of each round, the principle of this protocol is similar to one that has been presented in chapter 5 . This main difference is due to the extra assumption that strengthens the basic purely asynchronous computing model. The protocol of chapter 5 uses a timer-based assumption (it assumes an $\diamond 2t$-bisource) whereas the present one uses a time-free assumption. Each process p_i executes the protocol given by Figure 6.1. It is composed of a main task ($(T1)$), a decision task ($(T3)$) the aim of which is to allow a process to stop participating in the protocol when it decides. It implements some kind of reliable broadcast of the decision value (certified value). $T2[]$ is an array of tasks, each associated with a round r executed by process p_i. It is tasks $T2$ that implement the query-response mechanism of the coordination phase as explained in the following. The proposed protocol uses authentication to reduce the power of Byzantine processes. Indeed, a Byzantine process p can relay falsely a value it has received from some process q. If process q signs its message and process p cannot forge the signature of q then either p relays correctly the message of q or it does not relay it at all (the signed message received by p from q is the certificate it has to append to the message it uses to relay the message it received from q). In the latter p can still lie by saying that it received no value from q. Now suppose that p has to send to all processes the majority value it has received (the most frequent value among all the values it has received). The certificate will consist of the set of signed message it has received. By this mean any process can check whether the majority value sent by the process is sound. Note that, this does not prevent some process p from cheating. For example, if a Byzantine process p receives all of the sent messages (n messages, one from each process of the system), it can build two sets of $(n-t)$ messages that lead to two different most frequent values and then sent each of these two values to different processes.

Each process p_i manages a variable est_i to store its estimate of the decision value. In order to ensure the validity property, the protocol starts with an init phase (lines 01-01) to initialize the variable est_i. This phase consists of an all-to-all message exchange that allows to initializes the local variable est_i of a process p_i to a value it has received at least (n - 2t) times if any. Otherwise, est_i is set to v_i the value proposed by p_i. In the case where all correct processes propose the same value v, the only value that can be received at least $(n-2t)$ times is v and moreover any of $(n-t)$ received messages contains at least $(n-2t)$

times the value v. Consequently, all possible sets of $(n-t)$ received messages certify only v and no Byzantine process can introduce a wrong value as it will be discovered. If not all correct processes do propose the same value, it may happen that among the values received by a correct process p, no value is received $(n-2t)$ times or more. In such a case, process p keeps the value it proposes and can use the set of $(n-t)$ signed messages it received as a certificate to justify why it kept its own value. After the init phase, each process executes consecutive asynchronous rounds. Each round is composed of four communication phases and is coordinated by a predetermined process.

First phase of a round r (lines 05-07). Each process that starts a round (including the coordinator of the round) first sends its own estimate (with the associated certificate) to all processes. In a separate task (line 20), Each time a process receives a valid QUERY message (perhaps from itself) containing an estimate est, it sends a RESPONSE message to the sender. If the process that responds to a query message is the coordinator of the round to which is associated the query message, the value it sends in the RESPONSE message is the coordination value. If the process that responds is not the coordinator, it responds with any value as the role of such a message is only to define winning links. as the reader can find it in line 19-22, the value sent by the coordinator is the value contained in the first valid query message of the round it coordinates. In the main task at line 06, a process p_i waits for the response from pc (the coordinator of the round) or from $(n-t)$ responses from others processes. In the latter case, process p_i is sure that p_c is not the right winning process as its response is not winning. If a process receives a response from the coordinator then it keeps the value in a variable aux otherwise it sets aux to a default value $\perp$ (this value cannot be proposed).

RESPONSE(r, est) messages are sent by each process from another parallel task $T2[r]$ because the coordinator of round r could be stuck in previous rounds and if it does not respond quickly, the sender on the QUERY message may receive $(n-t)$ RESPONSE messages from others processes. There is one task T per round. When the coordinator receives the first valid QUERY message for a round it coordinates, it stores the included estimate in a local variable c_est_i. It is this value that the coordinator will send as all RESPONSE messages to the query messages associated with this round that it will receive (this allows a coordinator to coordinate a round with a certified value it has received even if it is itself lying far behind). The others RESPONSE messages sent by the others processes than the coordinator are only used to prevent processes from blocking while waiting (line 06) for the response of a faulty coordinator and the values carried by these messages are not used by processes.

If the current coordinator is a $\diamond 2t$-winning it has at least $2t$ privileged processes among which at least t are correct processes. Consequently, at least $(t+1)$ correct processes (the t correct processes and the coordinator itself) got the value v of the coordinator and thus set their variable aux to v $(v \neq \perp)$. If the current coordinator is not a $\diamond 2t$-winning process or if it is Byzantine, the three next phases allow correct processes to behave in a consistent way. The aim of the first phase is that if the coordinator is an $\diamond 2t$-winning process then at least $(t+1)$ correct process will get its value at the end of line (01).

Second phase of a round r (lines 08-10).

At the end of the first phase, if the current coordinator is an $\diamond 2t$-winning process then at least $(t+1)$ correct processes set their variable aux_i to the same non-$\perp$ value (the value

sent by the coordinator in RESPONSE messages). During the second phase, all correct processes relay, at line 09, either the value they received from the coordinator (with its certificate) or the default value $\perp$ if they received $(n-t)$ RESPONSE messages from others processes. Each process collects $(n-t)$ valid messages and stores the values in a set V_i (line 09).

At line 09, if the coordinator is correct only one value is valid and can be relayed. Moreover, if the coordinator is a $\diamond 2t$-winning process then any correct process p_i will get in its set V_i at least one copy of the value of the coordinator as among the $(t+1)$ copies sent by the $(t+1)$ correct processes that got the value of the coordinator a correct process cannot miss more than t copies (recall that a correct process collect $(n-t)$ valid messages). If the coordinator is not an $\diamond 2t$-winning process or if it is Byzantine, this phase has no particular effect. The aim of this second phase is that if the coordinator is an $\diamond 2t$-winning process then all the correct processes will get its value.

Third phase of a round r (lines 11-13).

This phase is a filter; it ensures that at the end of this phase, at most one non-$\hat{\text{U}}$ value can be kept in the *aux* variables in the situations where the coordinator is Byzantine. If the coordinator is correct, this is already the case. When the coordinator is Byzantine two different correct processes may have set their aux_i variables to different values. In this phase, each

process collects $(n-t)$ valid messages, the values of which are stored in a set V_i. If V_i carries only the same value v ($V_i = v$) then v is kept in aux_i otherwise aux_i is set to $\perp$. At the end of this phase, there is at most one certified value $v (v \neq \perp)$. This phase has no particular effect if the coordinator is correct. It ensures that eventhough the coordinator is Byzantine, at most one value is kept in the *aux* variables.

Fourth phase of a round r (lines 14-17).

This phase ensures that the Agreement property will never be violated. This prevention is done in the following way. If a correct process p_i decides v during this round then if some processes progress to the next round, then v is the only certified value. In this decision phase, a process p_i collects $(n-t)$ valid messages and store the values in V_i. If the set V_i of a process p_i contains a unique non_$\perp$ value v, p_i decides v. Indeed among the $(n-t)$ same values v received by p_i, at least $n-2t$ have been sent by correct processes. As $(n-t)+(n-2t) > n$ any set of $(n-t)$ valid signed messages of this phase includes at least one value v. Hence, all processes receive at least one value v (the other values could be v or $\perp$) and the only certified value for the next rounds is v. This means that during the next round (if any) no coordinator (whether correct or Byzantine) can send a valid value different from v.

If during the fourth phase, a process p_i receives only $\perp$ values, it is sure that no process can decide during this round and thus it can keep the value it has already stored in est_i (the certificate composed of the $(n-t)$ valid signed messages received during phase four containing $\perp$ values, allow p_i to keep its previous values est_i).

Before deciding (line 16), a process first sends to all other processes a signed message DEC that contains the decision value (and the associated certificate). When a process pi

receives a valid DEC message at line 23, it first relays is to all other processes and then decides (not all processes decide necessarily during the same round).

6.4 Conclusion

This chapter presented a time-free deterministic protocol that solves authenticated Byzantine Consensus in an asynchronous distributed system. The protocol assumes a weak additional assumption on message pattern where at least $2t$ communication links are eventually winning. These links connect the same correct process. In favorable setting, the proposed protocol can reach decision in only 5 communication steps and needs only $\Omega(n_2)$ messages in each step.

Function Consensus(v_i)

Init: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;

Task $T1$: % basic task %

——————————————————- init phase ——————————————————

(01) *send* INIT(v_i) to all;
(02) **wait until** (INIT messages received from at least $(n-t)$ distinct processes);
(03) **if** ($\exists v$: received at least $(n-2t)$ times) **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$ **endif**;

repeat forever
(04) $c \leftarrow (r_i mod n) + 1$; $r_i \leftarrow r_i + 1$;

——————————————————- round r_i ——————————————————

(05) *send* QUERY(r_i, est_i) to all;
(06) **wait until** (RESPONSE(r_i, est) received from p_c)
or (RESPONSE(r_i, est) received from $(n-t)$ distinct processes);
(07) **if** RESPONSE(r_i, est) received from p_c **then** $aux_i \leftarrow est$ **else** $aux_i \leftarrow \perp$;

(08) *send* RELAY(r_i, aux_i) to all;
(09) **wait until** (RELAY($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(10) **if** $(V_i - \{\perp\} = \{v\})$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(11) *send* FILT1(r_i, aux_i) to all;
(12) **wait until** (FILT1($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(13) **if** $(V_i = \{v\})$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

(14) *send* FILT2(r_i, aux_i) to all;
(15) **wait until** (FILT2($r_i, *$) received from at least $(n-t)$ distinct processes) **store values** in V_i;
(16) **case** $(V_i = \{v\})$ **then** *send* DEC(v) to all; **return**(v);
(17) $(V_i = \{v, \perp\})$ **then** $est_i \leftarrow v$;
(18) **endcase**;

———————————————————————————————————————

end repeat

Task $T2$[r]: % Query-response coordination task - There is one such task per round r %
(19) $c\_est_i \leftarrow \perp$
(20) **upon** *receipt* of QUERY(r, est) from p_j;
(21) **if** p_i is the coordinator of the round r and $c\_est_i \leftarrow \perp$ **then** $c\_est_i \leftarrow est$;
(22) *send* RESPONSE(r_i, c_est_i) to p_j

Task $T3$:
(23) **upon** *receipt* of DEC(est): *send* DEC(est) to all; **return**(est);

Figure 6.1: An Authenticated Byzantine Consensus Protocol With $\diamond 2t$-Winning

Bibliography

[1] [1] Aigner M. and Ziegler G., *Proofs from THE BOOK* (4th edition). Springer, 274 pages, 2010 (ISBN 978-3-642-00856-6).

[2] [2] Attiya H. and Welch J., *Distributed computing: fundamentals, simulations and advanced topics*, (2d Edition), Wiley-Interscience, 414 pages, 2004.

[3] [3] Ben-Or M., Another advantage of free choice: completely asynchronous agreement protocols. *Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing(PODC'83)*, ACM Press, pp. 27-30, 1983.

[4] [4] Berman P. and Garay J.A., Randomized distributed agreement revisited. *Proc. 33rd Annual Int'l Symposium on Fault-Tolerant Computing (FTCS'93)*, IEEE Computer Press, pp. 412-419, 1993.

[5] [5] Bracha G., An asynchronous $(n-1)/3$-resilient consensus protocol. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 154-162, 1984.

[6] [6] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143, 1987.

[7] [7] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840, 1985.

[8] [8] Cachin Ch., Kursawe K., and Shoup V., Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology*,18(3):219-246, 2005 (first version: PODC 2000).

[9] [9] Cachin Ch., Kursawe K., Petzold F., and Shoup V., Secure and efficient asynchronous broadcast protocols. *Proc. 21st Annual International Cryptology Conference CRYPTO'01*, Springer LNCS 2139, pp. 524–541, 2001.

[10] [10] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, 2011 (ISBN 978-3-642-15259-7).

[11] [11] Canetti R., and Rabin T., Fast asynchronous Byzantine agreement with optimal resilience, *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 42-51, 1993.

[12] [12] Chandra T. and Toueg S., Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225-267, 1996.

[13] [13] Correia M., Ferreira Neves N., and Verissimo P., From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82-96, 2006.

[14] [14] Dwork C., Lynch N., and Stockmeyer L., Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), 288-323, 1988.

[15] [15] Fischer M.J. and Lynch N., A lower bound for the time to ensure interactive consistency. *Information Processing Letters*, 14:183-186, 1982.

[16] [16] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, 1985.

[17] [17] Fischer M.J. and Merritt M., Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3): 239-247, 2003.

[18] [18] Friedman R., Mostéfaoui A., Rajsbaum S., and Raynal M., Distributed agreement problems and their connection with error-correcting codes. *IEEE Transactions on Computers*, 56(7):865-875, 2007.

[19] [19] Friedman R., Mostéfaoui A. and Raynal M., $\Diamond\mathcal{P}_{mute}$-based consensus for asynchronous Byzantine systems. *Parallel Processing Letters*, 15(1-2):162-182, 2005.

[20] [20] Friedman R., Mostéfaoui A., and Raynal M., Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46-56, 2005.

[21] [21] Goldwasser S., Pavlov E., and Vaikuntanathan V., Fault-tolerant distributed computing in full-information networks. *Proc. 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*,IEEE Computer Society, pp. 15-26, 2006.

[22] [22] Gray J. and Reuter A., *Transaction processing: concepts and techniques*, Morgan Kaufmann Pub., San Francisco (CA), 1045 pages, 1993, (ISBN 1-55860-190-2).

[23] [23] Kihlstrom K.P., Moser L.E. and Melliar-Smith P.M., Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16-35, 2003.

[24] [24] King V. and Saia J., Breaking the $O(n^2)$ bit barrier: scalable Byzantine agreement with an adaptive adversary. *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 420-429, 2011.

[25] [25] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401, 1982.

[26] [26] Liang G. and Vaidya N., Error-free multi-valued consensus with Byzantine failures. *Proc. 30th ACM Symposium on Principles of Distributed Computing (PODC'11)*, ACM Press, pp. 11-20, 2011.

[27] [27] Lynch N.A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, 1996 (ISBN 1-55860-384-4).

[28] [28] Lynch N.A., Merritt M., Weihl W.E., and Fekete A., *Atomic Transactions*. Morgan Kaufmann Pub., San Francisco (CA), 500 pages, 1994 (ISBN 1-55860-104-X).

[29] [29] Martin J.-Ph. and Alvisi L., Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202-215, 2006.

[30] [30] Milosevic Z., Hutle M., and Schiper A., On the reduction of atomic broadcast to consensus with Byzantine faults. *Proc. 30th IEEE Int'l Symposium on Reliable Distributed Systems (SRDS'11)*, IEEE Computer Press, pp. 235-244, 2011.

[31] [31] Mostéfaoui A., Moumen H., and Raynal M., Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. *Proc. 33th ACM Symposium on Principles of Distributed Computing (PODC'14)*, ACM Press, pp. 2-9, 2014.

[32] [32] Mostéfaoui A., Rajsbaum S., and Raynal M., Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922-954, 2003.

[33] [33] Mostéfaoui A. and Raynal M., Signature-free broadcast-based intrusion tolerance: never decide a Byzantine value. *Proc. 14th Int'l Conference On Principles Of Distributed Systems (OPODIS'10)*, Springer LNCS 6490, pp. 144-159, 2010.

[34] [34] Mostéfaoui A. and Raynal M., Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. *Proc. 22nd Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'15)*, Springer LNCS, 2015.

[35] [35] Patra A., Error-free multi-valued broadcast and Byzantine agreement with optimal communication complexity. *Proc. 15th Int'l Conference On Principles Of Distributed Systems (OPODIS'10)*, Springer LNCS 7109 pp. 34-49, 2011.

[36] [36] Pease M., R. Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234, 1980.

[37] [37] Rabin M., Randomized Byzantine generals. *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS'83)*, IEEE Computer Society Press, pp. 116-124, 1983.

[38] [38] Raynal M., *Communication and agreement abstractions for fault-tolerant asynchronous distributed systems*. Morgan & Claypool, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[39] [39] Raynal M., *Fault-tolerant agreement in synchronous message-passing systems*. Morgan & Claypool, 165 pages, 2010 (ISBN 978-1-60845-525-6).

[40] [40] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, 2013 (ISBN 978-3-642-32026-2).

[41] [41] Song Y.J. and van Renesse R., Bosco: one-step Byzantine asynchronous consensus. *Proc. 22th Symposium on Distributed Computing (DISC'08)*, Springer LNCS 5218, 438-450, 2008.

[42] [42] Srikanth T.K. and Toueg S., Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80-94, 1987.

[43] [43] Toueg S., Randomized Byzantine agreement. *Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, ACM Press, pp. 163-178, 1984.

[44] [44] Turpin R. and Coan B.A., Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18:73-76, 1984.